ROBOTIC FINGERSPELLING HAND FOR THE DEAF-BLIND

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Mechanical Engineering

by

Jerry Vin

November 2013

COMMITTEE MEMBERSHIP

TITLE:	Robotic Fingerspelling Hand for the Deaf-Blind
AUTHOR:	Jerry Vin
DATE SUBMITTED:	November 2013
COMMITTEE CHAIR:	Dr. Saeed Niku, Professor of Mechanical Engineering
COMMITTEE MEMBER:	Dr. John Ridgely, Professor of Mechanical Engineering
COMMITTEE MEMBER:	Dr. William Murray, Professor of Mechanical Engineering

ABSTRACT

Robotic Fingerspelling Hand for the Deaf-Blind

Jerry Vin

Because communication has always been difficult for people who are deaf-blind, The Smith-Kettlewell Eye Research Institute (SKERI), in conjunction with the California Polytechnic State University Mechanical Engineering department, has commissioned the design, construction, testing, and programming of a robotic hand capable of performing basic fingerspelling to help bridge the communication gap. The hand parts were modeled using SolidWorks and fabricated using an Objet rapid prototyper. Its fingers are actuated by 11 Maxon motors, and its wrist is actuated by 2 Hitec servo motors. The motors are controlled by Texas Instruments L293D motor driver chips, ATtiny2313 slave microcontroller chips programmed to act as motor controllers, and a master ATmega644p microcontroller. The master controller communicates with a computer over a USB cable to receive sentences typed by a sighted user. The master controller then translates each letter into its corresponding hand gesture in the American Manual Alphabet and instructs each motor controller to move each finger joint into the proper position.

Keywords: disabilities, deaf-blind, assistive technology, engineering, mechanical, robotics, robotic hands, mechatronics

ACKNOWLEDGMENTS

- Dr. Deborah Gilden of The Smith-Kettlewell Eye Research Institute for developing and sponsoring this project and for offering her constant support and encouragement.

- Dr. Saeed Niku for chairing the thesis committee and for offering me the chance to work on this amazing project.

- Dr. John Ridgely and Dr. William Murray for joining the thesis committee even while busy with many other projects and for offering engineering advice and support seemingly on demand.

- Terry Cooke for his helpful programming and design advice.

- Larry Coolidge for his patience and assistance with rapid prototyping

- Cal Poly Machine Shop technicians for their manufacturing support and advice

- All Cal Poly students who have expressed interest and fascination with the project

- Maxon Motors for their extremely generous donation of 25 motors to the project.

TABLE OF CONTENTS

LIST OF TABLES ix
LIST OF FIGURES x
LIST OF CODE BLOCKS xii
1 Introduction 1
1.1 Deaf-Blindness . . . 1
1.1.1 Definition . . . 1
1.1.2 Causes . . . 1
1.1.3 Statistics . . . 2
1.1.4 Challenges Resulting from Deaf-Blindness . . . 2
1.1.5 Deaf-Blind Communication . . . 4
1.2 Fingerspelling and Tactile Signing . . . 7
1.2.1 American Sign Language / American Manual Alphabet . . . 7
1.2.2 Project Implications . . . 8
2 Project Development 9
2.1 Project Overview . . . 9
2.1.1 Requirements and Design Specifications . . . 10
2.2 Robotic Hands and Other Communication Tools . . . 12
2.2.1 DeafBlind Communicator . . . 12
2.2.2 Advanced Robotic Hands . . . 13
2.2.3 Previous Robotic Finger-Spelling Hands . . . 15
2.2.4 Application to this Project . . . 17
2.3 Human Hands . . . 18
2.3.1 Parts . . . 18
2.3.2 Articulations . . . 19
2.3.3 Application to this Project . . . 19
3 Design and Construction 23
3.1 Mechanical System . . . 23
3.1.1 General Part Size and Shape . . . 23
3.1.2 Actuation . . . 24
3.1.3 Hand Articulations . . . 26
3.1.4 Forearm . . . 30
3.2 Electrical Power and Control System . . . 31
3.2.1 Electrical Layout . . . 31
3.2.2 Clock . . . 33
3.2.3 Master Microcontroller . . . 34
3.2.4 Slave Motor Controllers . . . 36
3.2.5 Motor Drivers . . . 37
3.2.6 Communication . . . 39
3.2.7 Power . . . 41

4 Programming and Software 43
4.1 Slave Motor Controller . . . 43
4.1.1 Communication and Data Task . . . 43
4.1.2 Encoder Reading . . . 45
4.1.3 Motor Output Task . . . 47
4.2 Master Microcontroller . . . 49
4.2.1 Communication . . . 49
4.2.2 User Interface Task . . . 50
4.2.3 Output Task . . . 51
5 Testing 58
5.1 Current Project Status . . . 58
5.1.1 Hardware . . . 58
5.1.2 Software . . . 58
5.2 Evaluation of Design Specifications . . . 59
5.2.1 Dimensional Specifications . . . 59
5.2.2 Weight . . . 59
5.2.3 Assembly and Disassembly . . . 59
5.2.4 Specifications Not Tested . . . 60
6 Conclusions 61
6.1 Conclusions . . . 61
6.2 Recommendations . . . 61
6.2.1 Mechanical Recommendations . . . 61
6.2.2 Electrical Recommendations . . . 64
6.2.3 Overall Project Structure and Scope . . . 65
BIBLIOGRAPHY 66
APPENDIX 68
A American Manual Alphabet 68
B Finger Configurations 69
C Parts List 72
D Electrical Diagrams 75
D.1 Clock Diagram . . . 75
D.2 Communication Diagram . . . 76
D.3 Power Diagram . . . 77
D.4 Slave Output Diagram . . . 78
E State-Transition Diagrams 79
E.1 Slave . . . 79
E.1.1 Motor Task . . . 79
E.1.2 Data Task . . . 80
E.2 Master . . . 81
E.2.1 Output Task . . . 81
E.2.2 User Interface Task . . . 82
F How to Operate the Hand 83

G Code 90
G.1 Slave 90
G.2 Master 110
G.3 Master Libraries from ME 405 Mechatronics 165

LIST OF TABLES

2.1 Design Requirements 11
2.2 Design Specifications 12

4.1 Slave Motor Controller Character Commands 44
4.2 Main Menu 50

B.1 Table of Finger Configurations 69

C.1 Table of Electrical Parts 72
C.2 Table of Mechanical Parts 73

LIST OF FIGURES

1.1 Helen Keller overcame her deaf-blind disabilities to become one of the world's most influential activists, authors, and lecturers. 5

2.1 Information Flow Diagram 9
2.2 Communication between a deaf-blind person (left) and a sighted person (right) using the DeafBlind Communicator system. 13
2.3 Shadow Robot Dextrous Hand System C6M 14
2.4 SKERI's Dexter (left) and Dexter II (right) 15
2.5 Gallaudet's Hand and RALPH 16
2.6 The most recent Robotic Finger-Spelling Hand thesis at Cal Poly . . 17
2.7 Bones of the human hand 18
2.8 The angle formed by the distal-intermediate joint (A) is equal to the angle formed by the intermediate-proximal joint (B) 20
2.9 The angle formed by the proximal-metacarpal (A), proximal-intermediate (B), and distal-intermediate (C) joints are all equal. 21
2.10 The index finger is the only finger allowed to wag. 22

3.1 Photograph of a Maxon motor used in the hand 24
3.2 Maxon motors inside the hand 25
3.3 Servo motors used in the wrist. A) HS 225 BB (twisting motion). B) HS 985 MG (knocking motion) 26
3.4 Distal-intermediate linkage on the pinky 27
3.5 Thumb motions. A) Unbent. B) Distal-proximal. C) Proximal-metacarpal. D) Metacarpal-wrist (stretch). E) Metacarpal-wrist (fold) 28
3.6 Finger motions. A) Unbent. B) Intermediate-proximal. C) Proximal-metacarpal (bending) D) Proximal-metacarpal (wagging) 28
3.7 Wrist parts disconnected from the forearm and hand 30
3.8 Forearm connected to the hand and wrist 31
3.9 Top of the electrical board 32
3.10 Bottom of the electrical board 32
3.11 Port locations for programming each microcontroller 33
3.12 Port locations to plug in motor cables 33
3.13 Location of all quartz crystal oscillators on the board 34
3.14 ATmega644p master microcontroller pinout 35
3.15 Location of the master microcontroller on the board 35
3.16 Location of the servo output ports on the board 36
3.17 ATtiny2313 slave motor controller pinout 36
3.18 Location of the ATtiny2313 slave motor controllers on the board . . . 37
3.19 L293D motor driver pinout 37
3.20 Location of the L293D motor drivers on the board 38
3.21 Location of the MOSFET controlled by the master microcontroller . . 38

3.22 Sparkfun USB to serial breakout board 39
3.23 CD74HC4076E multiplexer pinout 40
3.24 Communication components on the board. A) USB to serial breakout board output. B) Slave to master MUX. C) Master to Slave MUX. . 40
3.25 AC adapter (left) and battery (right) 41
3.26 Location of power components on the board. 5V voltage regulators for microcontrollers, 6V voltage regulator for servos, and screw terminals for input power 42

4.1 Encoder quadrature system for two incremental encoder channels . . 47
4.2 Stretch, curl, and clenched configurations common to all fingers . . . 52
4.3 Vertical clench and fold configurations used by the index and middle fingers 53
4.4 Normal, U, and cross configurations used only on the index finger . . 53
4.5 Thumb configurations. Flat up, folded up, folded in, folded out, stretched, curled 54
4.6 Servo PWM signal format 56

F.1 Shape of the USB Mini-B plug 83
F.2 Locating the Device Manager in the Start Menu 84
F.3 Identifying the COM number using the Device Manager (COM8 in this case) 85
F.4 PuTTY Security Warning 85
F.5 PuTTY Main Session Menu 86
F.6 PuTTY Terminal Menu 87
F.7 PuTTY Connection >Serial Menu 88
F.8 Main Menu for Operating the Hand 89

4.1 Checking to see if a command has been received 44
4.2 Example of send command usage 45
4.3 Loading gain and pre-set angular position data 45
4.4 Interrupt service routine for encoder reading 46
4.5 Motor output value calculations 48
4.6 Motor output commands 48
4.7 Master motor output example 55
4.8 Pull-cable motor example 55
4.9 Servo control example 57
G.1 Main File slave.cpp 90
G.2 Serial Header serial.h 100
G.3 Serial Class serial.cpp 102
G.4 Motor Header motor.h 105
G.5 Motor Class motor.cpp 107
G.6 Motor Preset Data angles.h 109
G.7 Main File master.cpp 110
G.8 Output Task Header task_output.h 113
G.9 Output Task Class task_output.cpp 116
G.10 User Interface Task Header task_user.h 140
G.11 User Interface Task Class task_user.cpp 143
G.12 Slave Picker Header slave_picker.h 156
G.13 Slave Picker Class slave_picker.cpp 158
G.14 Servo Header servo.h 161
G.15 Servo Class servo.cpp 162
G.16 State Transition Logic Timer Header stl_timer.h 165
G.17 State Transition Logic Timer Class stl_timer.cpp 170
G.18 State Transition Logic Task Header stl_task.h 181
G.19 State Transition Logic Task Class stl_task.cpp 188
G.20 Character Queue Header queue.h 198
G.21 Character Queue Class queue.cpp 204
G.22 Base Generic Serial Header base232.h 205
G.23 Base Generic Serial Class base232.cpp 208
G.24 Base Text Serial Header base_text_serial.h 213
G.25 Base Text Serial Class base_text_serial.cpp 220
G.26 RS232 Header rs232int.h 232
G.27 RS232 Class rs232int.cpp 235

Chapter 1: Introduction

While technology has advanced to the point where communication across the planet can occur instantly, one neglected segment of the world's population still has trouble communicating beyond the reach of their fingertips. This group is the community of deaf-blind individuals. Because their dual disability precludes both their vision and their hearing, deaf-blind individuals must rely on their sense of touch to interact not only with other people, but with their immediate surroundings as well.

1.1 Deaf-Blindness

1.1.1 Definition

As defined by the United States Government for the purposes of special education, deaf-blindness is

> *"concomitant hearing and visual impairments, the combination of which causes such severe communication and other developmental and educational needs that they cannot be accommodated in special education programs solely for children with deafness or children with blindness."*

Extending this definition to encompass adults as well as children, deaf-blindness can refer to any combination of hearing and vision loss severe enough that assistive tools and resources designed to help people who are solely deaf or solely blind are not enough to compensate. This definition includes, but does not necessarily imply, total loss of both hearing and vision.[1]

1.1.2 Causes

Some children may be born deaf-blind, while others may develop one or both impairments later in life through either accident or illness.

- **Syndromes:** Down's Syndrome, Usher Syndrome, Patau Syndrome

- **Congenital Causes:** CHARGE Association, Hydrocephaly, Microcephaly, Fetal alcohol syndrome or other maternal drug abuse, Prematurity, AIDS, Rubella, Toxoplasmosis, Herpes, Syphilis

- **Post-natal Causes:** Asphyxia, Head trauma, Stroke, Encephalitis, Meningitis, Injuries that damage sight or hearing

1.1.3 Statistics

Several attempts to measure the impact of deaf-blindness have been conducted in the United States. A 2008 resolution by the National Association of Regulatory Utility Commissioners estimates that 70,000 to 100,000 Americans are both deaf-blind and cut off from basic telecommunications services due to the high costs of devices designed to assist people who are deaf-blind.[2]

The National Consortium on Deaf-Blindness (NCDB) annually attempts to obtain a more accurate count of American children and youth that suffer from deaf-blindness by obtaining statistics from child-care services. Their 2010 census estimates that a total of 9,320 infants, children, and young adults in the U.S. suffer from deaf-blindness in some form. Nearly 90% of the students counted also suffer from one or more additional disabilities[3].

1.1.4 Challenges Resulting from Deaf-Blindness

What makes deaf-blindness such a crippling disability is the severe deprivation of the level of sensory input that a deaf-blind person faces. Blind persons rely heavily on their hearing (speech, audiobooks, etc) to compensate for their lack of sight. Deaf persons rely heavily on their eyesight (sign language, fingerspelling, printed text, etc) to compensate for their lack of hearing. Impairments in both senses make interacting with the surrounding world incredibly difficult.

While some deaf-blind individuals can experience some usable vision or hearing, the main sense that can be relied upon for communication for all severities of

deaf-blindness is the sense of touch. Unfortunately this limits the means of communication primarily to the fingertips. Communication becomes an enormous challenge, especially when the majority of interactions occur with people who are not deaf-blind and have difficulty understanding the challenges that deaf-blind persons face.

Visual and audible cues that are often taken for granted are not perceived by the deaf-blind. Consistent failures of people with fully-functioning vision and hearing to help bridge the overwhelming gap of understanding between a deaf-blind person and his or her surroundings can often cause the person to develop behavioral and emotional problems that result from repeated frustration with and crippling fear of the surrounding world[1].

Because of communication difficulties, deaf-blind children do not respond or behave in the same manner as children with sight or hearing. Much of linguistic development in the average child comes from pattern recognition using visual and audible stimuli. Deaf-blind children miss out on this stimuli and therefore suffer from slower cognitive and social development due to a lack of incoming sensory exposure to the world. Even deaf-blind children with normal cognitive potential may be misdiagnosed as mentally handicapped due to a lack of communication and information during testing. Some are even unfortunate enough to end up in neglected sheltered groups and risk growing up to become severely emotionally disturbed and intellectually limited adults as a result of daily frustration with the surrounding world[4].

Deaf-blindness affects more than just deaf-blind persons. It affects people closest to them as well. The latter must learn to be engaged enough to communicate constantly and give enough feedback about the surrounding world to make sure the person is not isolated. Traditional visual body language gestures and cues as well as spoken words are no longer effective and must be replaced with

concepts that can be communicated through touch.

Family members, teachers, and caregivers must also have the patience and interactive mindset to be receptive to responses from the deaf-blind person, especially during child development stages. Much like most other children learn from both listening to conversation and speaking themselves, deaf-blind children must learn from incoming and outgoing communication as well. The major differences are the more arduous means through which this communication is expressed. Subtle gestures made by the deaf-blind person can represent important signals of affirmation, dissatisfaction, specific needs, pleasure, pain, etc. Failure to pick up on these signals leaves the deaf-blind communicators helpless[1].

Unfortunately parents of deaf-blind children frequently have little or no information and few support services to help with the raising of a deaf-blind child. Behavioral and social problems that result from the child's frustration are often directed towards caregivers, who are the child's primary connection to the world[4].

1.1.5 Deaf-Blind Communication

Teaching a deaf-blind person to communicate is the most effective way to ease the environmental adaptation process. Communication not only allows a person to express his needs and thoughts but also increases the rate of learning exponentially. Helping a person gain literacy opens up the following avenues and more:

- **Information consumption** through books, publications, instruction manuals, directories, and the internet
- **Memory and financial organization and supplementation** through calendars, lists, schedules, notes, and transcriptions
- **Self-expression** through writing journals and literary works
- **Entertainment** through literary works and the internet

- **Relationship building** through letters, notes, announcements, and the internet
- **Spatial awareness and identification** through labels, signs, maps, and directories

While a deaf-blind person has a much more difficult time integrating these activities into daily life, the mere ability provided by literacy to practice, struggle, and adapt to each activity drastically increases the potential for the person to adapt to surroundings. The ability to communicate through language allows a deaf-blind person to learn and therefore gives them the best opportunity to lead an independent and productive life[5].

Figure 1.1: Helen Keller overcame her deaf-blind disabilities to become one of the world's most influential activists, authors, and lecturers.

The most famous example of this is Helen Keller (pictured in Figure 1.1, who overcame her deaf-blind disabilities to become a distinguished author, political activist, and lecturer. Like Keller, many deaf-blind individuals today have enormous

intellectual potential, but unlike Keller, are held back by their inability to communicate.

Aside from simple touch cues and gestures, some communication methods used by people with severe deaf-blind disabilities include[1]:

- **Object symbols:** Association of actual objects as symbols for concepts and activities (such as a ball used to represent playtime)
- **Tadoma:** A method of "tactile lipreading" sometimes used in conjunction with other techniques for reinforcement.
- **Sign language:** Hand and body language gestures used to represent words, phrases, and meanings.
- **Fingerspelling:** Hand and finger gestures used to represent the different letters of the alphabet. Often used to supplement sign language to spell out proper nouns and other words that do not have associated sign language gestures.
- **Braille:** Raised dots that represent letters. Used both as print and electromechanical materials for the blind, the tactile nature of braille makes it accessible to the deaf-blind community as well.

While electromechanical braille devices do exist, braille is not widely known in the deaf-blind community. A common cause of deaf-blindness is Usher's Syndrome. People with Usher's Syndrome typically experience deafness from birth but do not experience vision loss until the late teens or early 20s. Often Usher's Syndrome is not diagnosed until the vision loss begins to take effect. Education up until this point is geared towards deafness: mainly sign-language and fingerspelling, and not braille. Usher's Syndrome patients often put off learning braille out of frustration or denial even after diagnosis, because it can be considered a final admission of

blindness. Finding braille teachers for deaf adults can also be difficult. These facts coupled with the fact that fingerspelling is a much more organic method of communication that requires only hands and no other tools makes fingerspelling a far more widespread communication technique[6].

1.2 Fingerspelling and Tactile Signing

Fingerspelling is the spelling of words, letter by letter, using gestures made using the human hand. Sighted persons who use fingerspelling use it in conjunction with sign language to spell out words that do not have a corresponding sign language gesture (or words whose sign language gestures are unknown by the user) such as proper nouns[7].

1.2.1 American Sign Language / American Manual Alphabet

Fingerspelling in American Sign Language consists of 36 one-handed letter and number gestures that make up the American Manual Alphabet. This is the version of fingerspelling that will be used in this project. An illustration of fingerspelling gestures in the American Manual Alphabet is located in Appendix A.

For most of the letters and numbers, fingerspelling gestures are made with the palm facing the reader. In deaf-blind fingerspelling, gestures are made in a similar manner, but with the reader's hand draped over the back of the hand near the fingers to feel changes in the different letters. Since some of the letters look similar, context is usually very important for distinguishing between characters such as the letter V and the number two[8].

American Manual Alphabet fingerspelling is not the only form of fingerspelling that exists or is commonly used. Other languages exist that use different letters, different gestures, or even multiple hands. While these different manual alphabets can be applied to more complex robotic hands, this particular project will not attempt to incorporate them because the American Manual Alphabet is by far the

most common form of fingerspelling for the intended audience of this project.

1.2.2 Project Implications

Understanding the American Manual Alphabet allows the simplification of the eventual final robot hand design. Because the hand will only be used for fingerspelling, not all of the different motions of the human hand are required. By simplifying the design to incorporate only the motions and degrees of freedom required for fingerspelling, the number of actuators involved in the design can be drastically reduced. This will help reduce the hand's complexity, weight, and cost. It will also allow more freedom to move around certain features within the limited volume and shape that would be allowed in order to make the hand look realistic. More information about this process is presented in Section 2.3.

Chapter 2: Project Development

2.1 Project Overview

The overall goal of this project is to build a functioning robotic hand capable of fingerspelling all letters and numbers in the American Manual Alphabet. This project is not necessarily intended for manufacturing and is more of a feasibility exploration and proof of concept for the idea using parts built by a rapid prototyper.

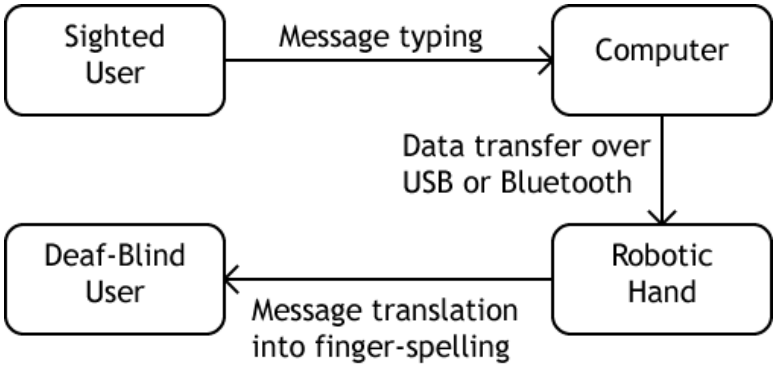


Figure 2.1: Information Flow Diagram

The motivations for this project are to help bridge the communication gap between deaf-blind fingerspellers and anyone unfamiliar with fingerspelling. This project is intended to supplement the efforts of fingerspelling interpreters and not necessarily replace them, especially since tactile fingerspelling is a physically arduous task. Initial versions of a robotic fingerspelling hand may be best used as a teaching tool. The device would also provide computer access to deaf-blind fingerspellers if interfaced properly through software. Future polished versions may be able to be manufactured and sold at a low enough cost that either deaf-blind individuals or third-party payers can afford one for regular communication use.

The basic flow of information is shown in Figure 2.1. First a sighted user types a message into a serial prompt on a computer. The computer then sends the

message over USB to the robotic hand. Then the hand fingerspells the message letter-by-letter while the deaf-blind "reader" interprets the hand's configurations.

This project has been pursued before for different senior projects and master's theses at Cal Poly. All editions of the project have been sponsored by the Smith-Kettlewell Eye Research Institute (SKERI).

2.1.1 Requirements and Design Specifications

After consultation with advisors and SKERI's Principal Investigator and project contact, a list of general requirements was formed. These requirements are presented in Table 2.1.

Table 2.1: Design Requirements

Requirement	Description
Accurate	The hand must be able to properly form all letters and numbers.
Quick	The hand should form all characters in roughly the same amount of time that a human hand would (roughly 2 letters per second).
Lifelike	The hand should resemble a human hand (size, shape, and texture) as much as possible.
Durable	Since the hand is used for tactile applications, it should be able to withstand the forces applied by a person while reading from the fingers.
Portable	The hand must be small and light enough to be transportable.
Easy to assemble and repair	The hand should be easy to assemble and disassemble if parts need to be replaced. Parts should be easy to access.
Easy to use	The computer program should be intuitive enough for a person with basic computer skills to use.
Safe	Deaf-blind users cannot process visual safety cues. No parts of the hand should pinch, burn, or shock the users.
Relatively inexpensive	The development budget for this project is $1500.

Each requirement is mapped to at least one quantifiable and measurable parameter to be evaluated. These parameters are the design specifications listed in Table 2.2.

Table 2.2: Design Specifications

Specification	Value	Unit	Tol.	Diff.	Test	Req.
Accuracy	90	%	± 10	Med	Touch Test	Accurate
Letter Formation Rate	2	Letters /sec	± 1	Hard	Timing	Quick
Max Palm Thickness	30	mm	Max	Med	Measure	Lifelike
Dev. from Average Limb Size	25	%	± 10	Med	Measure	Lifelike
Num. Broken Parts	0	Broken Parts	1	Med	General Use	Durable
Weight	10	lbs	± 1	Med	Scale	Portable, Safe
Assembly Time	20	Min	± 5	Med	Timed	Assembly
Dissassembly Time	20	Min	± 5	Med	Timed	Repair
Pinches, Burns, Shocks	0	Injuries	0	Hard	General Use	Safe
Cost	1500	$	Max	Med	Tracking	Inexpensive

2.2 Robotic Hands and Other Communication Tools

2.2.1 DeafBlind Communicator

One of the more common assistive devices for deaf-blind people is the DeafBlind Communicator system. The DeafBlind Communicator is a system of two handheld devices designed to allow deaf-blind users to communicate with sighted people. The

main unit is the DB BrailleNote, which consists of a Braille keyboard and special deaf-blind software. The second unit is the DB-Phone, which consists of a cell phone with a visual display and a QWERTY keyboard. These devices communicate over Bluetooth allowing a deaf-blind person to hand the DB-Phone to a sighted person and send a message to the DB-Phone using the DB BrailleNote. The two users can then carry on a conversation with each other using their respective keypads.

Figure 2.2: Communication between a deaf-blind person (left) and a sighted person (right) using the DeafBlind Communicator system.

The DeafBlind Communicator is also very versatile with connections to other devices and protocols. With the installation of a SIM card, the DeafBlind Communicator can send SMS text messages to other cell phone users. The DBC is also compatible with the TTY protocol and relay services. DBC devices can also communicate with nearby DBC devices and with personal computers.

The main drawbacks are the cost of the system (roughly $6,300) and the fact that many deaf-blind persons do not know how to read braille[9].

2.2.2 Advanced Robotic Hands

While robotic hands are quite advanced from a technological perspective, many different types either exist or are currently in development. Many existing robotic hands are even capable of performing the fingerspelling tasks required for this project. The unfortunate drawback for most of those is that they are designed to act

as gripping hands rather than gesturing hands. As a result they have many different features not needed for this application. All these extra features come at extra cost.

Figure 2.3: Shadow Robot Dextrous Hand System C6M

Two good examples are the Dextrous Hand System C6M (shown in Figure 2.3) and C6 developed by Shadow Robot Company. These machines are capable of 24 different motions with 20 degrees of freedom with built-in touch and force sensors. This is much more than is necessary for fingerspelling. The cost of these hands are £115,000 and £75,000 ($190,000 and $120,000) respectively.

Most robotic hands that have been designed for practical applications have been designed for use as gripping hands. Designing a hand to grip and pick up objects is not the purpose of this project. Few hands have been designed with the specific goal of forming accurate hand shapes for use in fingerspelling.

2.2.3 Previous Robotic Finger-Spelling Hands

As mentioned previously, this current project is another iteration of an ongoing string of attempts to build a robotic finger-spelling hand. Prior versions of the hand were moderately successful to very successful.

The very first robotic finger-spelling hand was built by the Southwest Research Institute (SWRI) in San Antonio, Texas in 1977. It consisted of a keyboard connected directly to the hand by electrical circuitry. This project proved that a robotic fingerspelling hand was feasible but had problems with forming all of the letters and with speed.

Figure 2.4: SKERI's Dexter (left) and Dexter II (right)

The Smith-Kettlewell Eye Research Foundation then sponsored a project at Stanford University to build a new hand, called Dexter (pictured in Figure 2.4), in 1985. Dexter was an aluminum hand attached to a box that contained all of its actuators. Finger and thumb motions were actuated by drive cables pulled by pneumatic cylinders in the box. All of the fingers joints were independently controlled. Dexter was able to form most letters but had to return to a neutral position between each letter.

Dexter II was built in 1988 and was approximately 10% of the size of the original. This version used DC servo motors to pull drive cables and could form approximately four letters per second. One major improvement highlighted by

testing was the consistency of Dexter II, which allowed deaf-blind readers of the hand to adapt to the hand better.

Figure 2.5: Gallaudet's Hand (left) and RALPH (right)

A similar hand was built by Gallaudet University in 1992. Testing of this hand revealed results of roughly 70% interpretation accuracy by deaf-blind testers. A photograph of this hand is shown in Figure 2.5.

The Department of Veteran Affairs Rehabilitation Research and Development Center built a fourth-generation hand named RALPH (pictured in Figure 2.5). This hand used a transmission system built from mechanical linkages rather than cables. RALPH was able to achieve 75% accuracy during tests and even greater accuracy after the failed sentences were repeated[6].

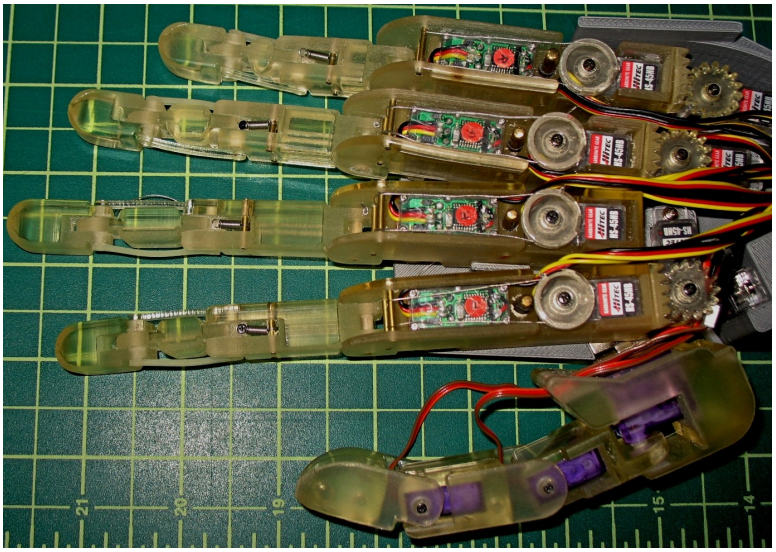


Figure 2.6: The most recent Robotic Finger-Spelling Hand thesis at Cal Poly

Several senior projects and at least one master's thesis project (2009) for robotic finger-spelling hand construction was assigned by SKERI to Cal Poly students. One senior project was a solenoid-actuated and cable-driven hand made from aluminum[10]. This design was very large and bulky. The most recent master's thesis project (shown in Figure 2.6) was another servo-actuated cable-driven hand that was able to condense the size of the hand by using small hobby servos[11].

2.2.4 Application to this Project

The goal of this project is to help bring the latest designs of the robotic fingerspelling hand closer to a manufacturable and marketable state. While previous designs had their own merits, this version of the hand should be much more realistic and should be able to achieve that realism at a lower cost. Another desired improvement over the most recent hand is an increase in the letter formation speed.

2.3 Human Hands

The human hand is one of the most complicated organs in all of nature. With 27 bones and many different motions, the human hand is capable of a wide combination of shapes and gestures. This versatility and complexity is what makes the hand capable of making the numerous gestures needed for fingerspelling. Unfortunately the same traits make mechanical replication a very difficult challenge to meet.

2.3.1 Parts

The human hand consists of five digits connected to a palm. This palm is connected to the human forearm by the wrist. The five digits are the thumb and index, middle, ring, and pinky fingers.

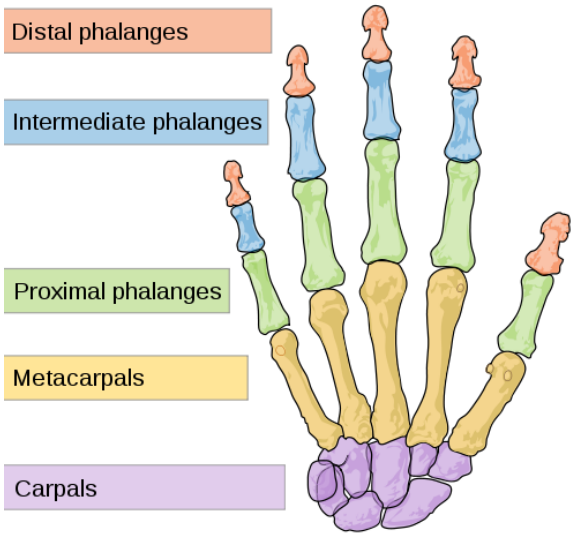


Figure 2.7: Bones of the human hand

Each of the index, middle, ring, and pinky finger digits contains three bones: the distal, intermediate, and proximal phalanges. The thumb contains only distal and proximal phalanges. Metacarpal bones stretch across the palm to connect proximal phalanges with the carpal bones in the wrist[12].

2.3.2 Articulations

With so many joints, the human hand is capable of many different motions or "articulations":

- **interphalangeal articulations:** bending of the hinge joints in the fingers (distal-intermediate and intermediate-proximal)
- **metacarpophalangeal articulations:** motions of the joints where the proximal and metacarpal bones connect in each finger (also where the metacarpal thumb joint meets the wrist)
- **intercarpal and wrist motions:** motions of the joints where the wrist and forearm connect and where the carpal bones in the wrist meet the metacarpal bones in the palm[12]

2.3.3 Application to this Project

The most important motions in the human hand are the ones made by the digits and wrist. The metacarpal bones can move slightly to shape the palm in different ways, but these motions are not critical for making the gestures required to fingerspell. The only critical metacarpal bone is in the thumb. This project will attempt to replicate the distal, intermediate, and proximal phalanges of the human hand while modeling the palm and wrist as single components rather than a collection of different bones.

While single axis rotations like the interphalangeal articulations can easily be replicated mechanically with a pin joint, the metacarpophalangeal articulations, intercarpal articulations, and wrist motions all occur about multiple axes of rotation. The easiest way to replicate and control multiaxis motions mechanically would be to split up the motions into multiple joints.

Because space was limited inside the hand, eliminating any unnecessary motors would help use space more efficiently. All the different fingerspelling gestures were

analyzed, and joint motions that were either unnecessary for fingerspelling or dependent on other motions were eliminated. These decisions were verified with the Cal Poly Disability Resource Center to make sure critical motions were not eliminated.

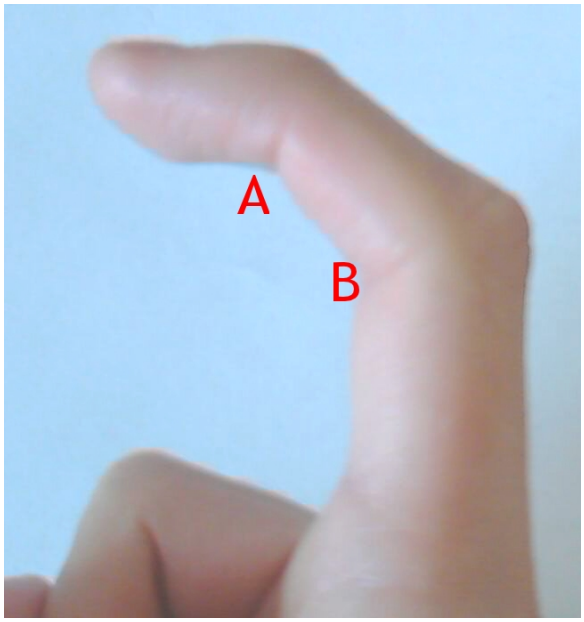


Figure 2.8: The angle formed by the distal-intermediate joint (A) is equal to the angle formed by the intermediate-proximal joint (B)

Interphalangeal motions for the distal-intermediate joints on the index, middle, ring, and pinky fingers are all roughly equal to the motions for the intermediate-proximal joints on the same fingers as shown in Figure 2.8. Most humans are not capable of moving just the distal-intermediate joint without moving the intermediate-proximal joint. Bending of one joint requires bending of the other joint. This relationship allows all four distal-intermediate joints to be mechanically linked to the intermediate-proximal joints on the same finger so only one of them (the intermediate-proximal joint since it is closer to the actuators) would have to be independently controlled.

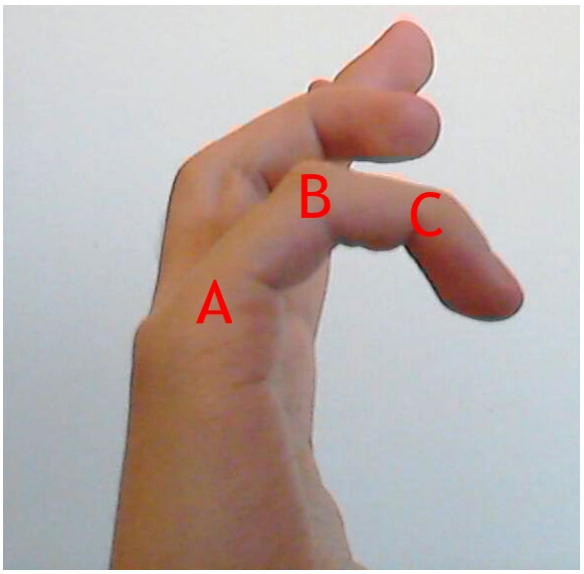


Figure 2.9: The angle formed by the proximal-metacarpal (A), proximal-intermediate (B), and distal-intermediate (C) joints are all equal.

For the purposes of fingerspelling, the ring and pinky fingers only have to curl and extend to close and open the joints. No fingerspelling gesture requires any of the joints in those fingers to curl while other joints in the same fingers remain straight. Therefore the intermediate-proximal interphalangeal motions can be linked to the proximal-metacarpal metacarpopholangeal motions in each of those two fingers. The distal-intermediate, intermediate-proximal, and proximal-metacarpal joints should all bend simultaneously with roughly the same joint angles as shown in Figure 2.9.

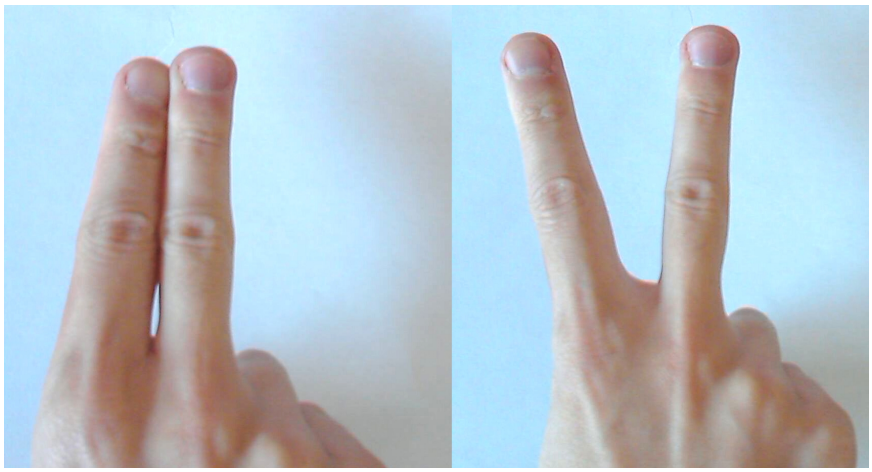


Figure 2.10: The index finger is the only finger allowed to wag.

For the purposes of fingerspelling, the metacarpopholangeal wagging motion of the proximal-metacarpal joints in the ring and pinky fingers can be eliminated as well. In a normal human hand these motions either move the fingers apart or bring them closer together. This motion is still somewhat necessary in the index and middle fingers to distinguish between the letters U and V or to cross the fingers to form the letter R, but not required in the ring and pinky fingers. Simply moving the index finger from side to side is enough to approximate this motion as shown in Figure 2.10.

There are three different motions in the wrist. The motion required for a standard hand wave can be eliminated, because it usually places stress on the wrist when used for fingerspelling. Any gestures that need that particular motion can be easily approximated using other motions. This has been confirmed through interviews with the Cal Poly Disability Resource Center[8].

This leaves 11 independent motions in the fingers and 2 independent motions in the wrist that require control.

Chapter 3: Design and Construction

The design for the robotic fingerspelling hand was developed over a period of roughly two years. During this time period several designs were conceived and then subsequently modified to arrive at the final design. From a high-level perspective, the entire system should be able to collect an input from the computer, send instructions to the hand, process those instructions, and convert those instructions into joint motions.

3.1 Mechanical System

3.1.1 General Part Size and Shape

Coming up with a general idea for size and shape of each component was not difficult. The requirements for the project asked for the final hand to be as lifelike as possible. The best source for measurements of a "lifelike" human hand were measurements from an actual human hand. Parts were modeled in SolidWorks and printed by a rapid prototyper.

The intermediate and proximal phalanges were designed to have hollow, round cross-sections. The roundness provided realism, while the hollow nature allows cables and wires to be run through the fingers rather than outside them. Shape complexity was only a small issue since most components would be printed on the rapid prototyper.

Ideally the parts should match the same dimensions of an actual human hand, but variance in human hand sizes provide some room for adjustment. Some dimensions had to be increased later to fit necessary components. Certain dimensions that do not interfere with finger-spelling, such as the thickness of the palm, were more flexible than others.

There are 27 parts that make up the hand (not including actuation

components):

- 15 finger and thumb phalanges
- 5 knuckles
- Bottom, middle, and top layers of the palm
- A textured cover of the bottom of the palm to hide cables
- 2 wrist parts
- A PVC pipe for the forearm to hold all electrical components

3.1.2 Actuation

Several different types of actuators were considered for this project including DC motors, servos, solenoids, pneumatic actuators, and hydraulic actuators. Most of these choices were based on actuators used in previous robotic hands. Most of these were ruled out due to size or weight. DC motors were deemed most appropriate since they came in relatively small packages and could be controlled using pulse width modulation.

Figure 3.1: Photograph of a Maxon motor used in the hand

The motors chosen for the project were donated by Maxon Motors. They are compact enough to fit inside a hand (roughly 5 cm long and 1 cm in diameter) and

came prepackaged with planetary gearheads and incremental encoders. A photograph of one of these motors is shown in Figure 3.1.

The motors were located in the palm to minimize the distance over which actuation would have to be transmitted. Due to space limitations, the best method of transmitting actuation from the motors to the actual finger parts was through drive cables. The motors were grouped together with the motor drive axes parallel to each other to use space efficiently. Figure 3.2 shows the motors inside the hand with cables connected.

Figure 3.2: Maxon motors inside the hand

The bidirectional transmission cables wrap around the rapid-prototyped pulley heads on the motors and are routed to the fingers over metal posts that help keep the different cables from interfering with each other. The drive cables are cut from 15 lb monofilament multipurpose line. The ends of each cable are looped through

1.3 mm (inside diameter) crimp tubes, tied, and crimped to prevent tension loss.

The Maxon motors did not have enough torque to control the wrists, so Hitec servo motors were selected to control the two wrist joints. The servo horns slide into the wrist parts and drive them directly. These servos are pictured in Figure 3.3

Figure 3.3: Servo motors used in the wrist. A) HS 225 BB (twisting motion). B) HS 985 MG (knocking motion)

3.1.3 Hand Articulations

All joints in the fingers are pinned joints with the exception of one of the metacarpal-wrist joints in the thumb and the two wrist motions. These three joints are directly attached to the motors that drive them. The following breakdown details all the motions that the robotic finger-spelling hand attempts to model.

Distal-Intermediate Motions

The distal-intermediate motions in the index, middle, ring, and pinky fingers are all dependent motions that don't require motor control. These joints are controlled by a mechanical linkage made from soft metal rod that connects the distal phalanx to the proximal phalanx.

Figure 3.4: Distal-intermediate linkage on the pinky

Figure 3.4 shows a photograph of this linkage as used on the pinky. This linkage ensures that the angular motion between the distal and metacarpal phalanges is equal to the angular motion between the metacarpal and proximal phalanges.

Thumb

Four independent motions are needed to control the thumb:

- Interphalangeal articulation of the distal-proximal joint
- Interphalangeal articulation of the proximal-metacarpal joint
- Two metacarpophalangeal motions at the metacarpal-wrist joint

These motions are shown in Figure 3.5.

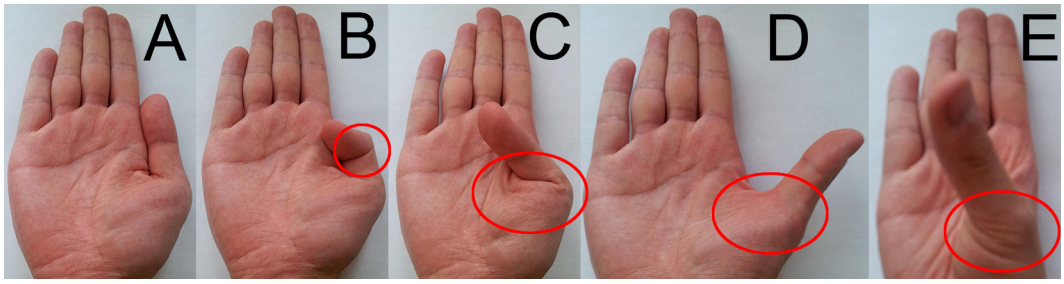


Figure 3.5: Thumb motions. A) Unbent. B) Distal-proximal. C) Proximal-metacarpal. D) Metacarpal-wrist (stretch). E) Metacarpal-wrist (fold)

Three of these motions are cable-driven. The metacarpophalangeal folding motion of the thumb is a direct drive motion.

Index Finger

Three independent motions are needed to control the index finger. The joints about which these motions occur are shown in Figure 3.6.

- Interphalangeal articulation of the intermediate-proximal joint
- Two metacarpophalangeal motions at the proximal-metacarpal joint

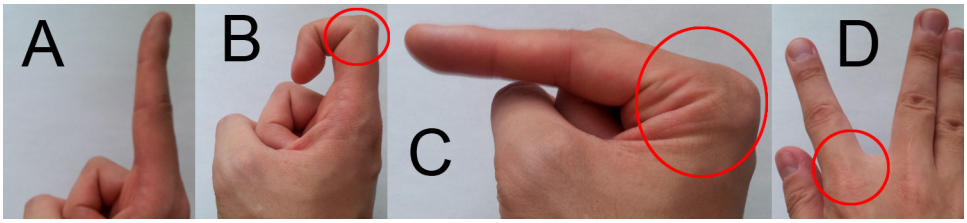


Figure 3.6: Finger motions. A) Unbent. B) Intermediate-proximal. C) Proximal-metacarpal (bending) D) Proximal-metacarpal (wagging)

All three of these motions are cable-driven. The wagging motion is an on-off mechanism that does not require accurate position control. Turning the motor on pulls a cable that pulls the index finger away from the middle finger. The physical

stop built into the design prevents the finger from moving too far away. Turning the motor off allows the tension in the cables to pull the index finger back towards the middle finger without having to reverse the motor.

Middle Finger

Two independent motions are needed to control the middle finger.

- Interphalangeal articulation of the intermediate-proximal joint
- Metacarpophalangeal motions at the proximal-metacarpal joint

These motions occur about joints B and C respectively from Figure 3.6.

Ring and Pinky Fingers

One independent motion is needed to control each of the ring and pinky fingers. The two joints that are independently controlled in the index and middle fingers are dependent on each other in the ring and pinky fingers and are controlled by the same motor instead of separate motors. These motions occur about joint C from Figure 3.6.

Each finger has both joints controlled by one single motor. The joints are structured very similar to the index and middle fingers. The main difference is that the cables from the metacarpal-proximal motion and the metacarpophalangeal hinge motion on the ring and pinky fingers are threaded through the same motor so that each joint turns at roughly the same angle. This causes the ring and pinky fingers to curl when these motors are activated. The ring and pinky fingers are not capable of maintaining one joint straight while bending the other.

Wrist

Two motions occur in the wrist. These motions are

- Radioulnar motions (twisting about vertical axis)

- Palmar flexion and extension (door knocking motion about horizontal axis)

Figure 3.7 shows the wrist parts separated from the rest of the hand. The head of the HS 985 MG servo motor connects with a servo horn embedded in the lower portion of the palm. Underneath the HS 985 MG servo is another servo horn that is screwed into the HS 225 BB servo motor below to provide the twisting motion about the vertical axis.

Figure 3.7: Wrist parts disconnected from the forearm and hand

3.1.4 Forearm

The forearm consists of a 2.5 inch PVC pipe with a bell on the upper end to mate with the rapid-prototyped part containing the lowest wrist joint (vertical axis twisting). It holds the electronics board and all the electrical cables from the motors.

Figure 3.8: Forearm connected to the hand and wrist

3.2 Electrical Power and Control System

3.2.1 Electrical Layout

Most of the electrical components are located on a single dual-layer printed circuit board. The board was designed using Cadsoft Eagle and manufactured by OSH Park. The board shape is designed specifically to fit in the forearm. Electrical diagrams detailing the connections between the major components of the board are included in Appendix D.

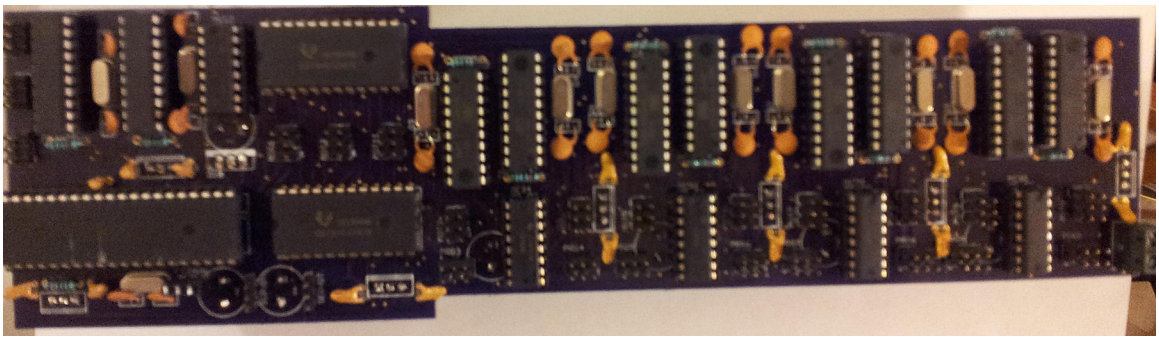


Figure 3.9: Top of the electrical board

A photograph of the top of the board, where most of the electrical components can be found, is shown in Figure 3.9. The bottom of the board where the MOSFET, voltage regulators, and 1000uF capacitors are located, is shown in Figure 3.10.

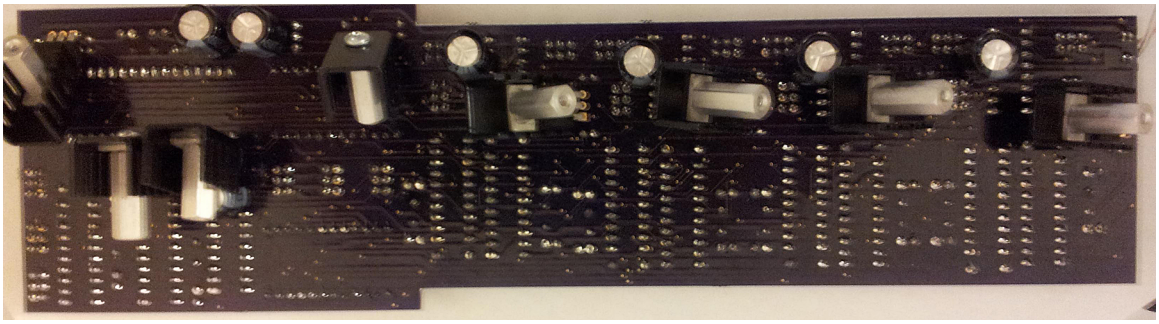


Figure 3.10: Bottom of the electrical board

Included on the board are AVR ISP 6-pin male headers for programming each microcontroller and 6-pin male headers for each motor output. Figure 3.11 shows the location of the programming ports on the board while Figure 3.12 shows the location of every motor output port.

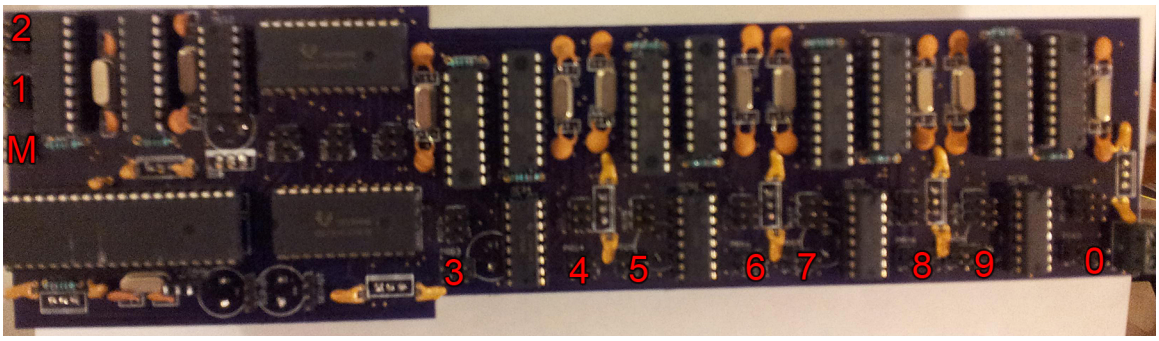


Figure 3.11: Port locations for programming each microcontroller

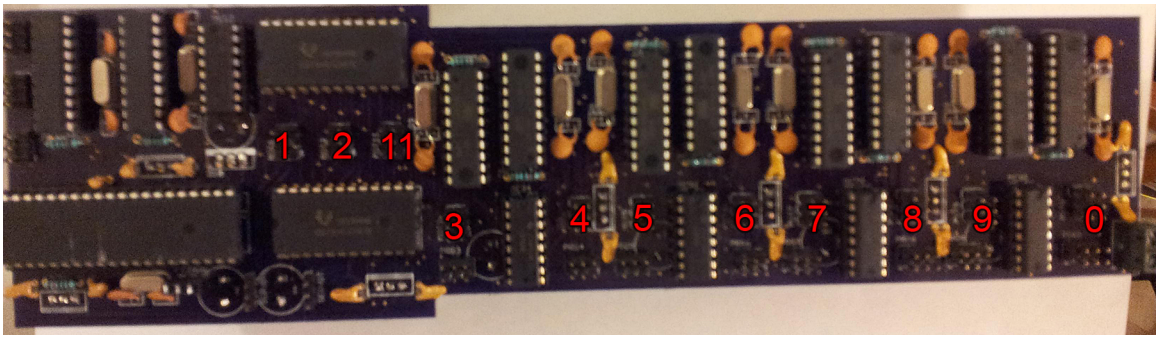


Figure 3.12: Port locations to plug in motor cables

3.2.2 Clock

All microcontrollers run at 20 MHz using an external Vishay quartz crystal oscillator. The output pins on each oscillator are connected to the XTAL1 and XTAL2 pins on the corresponding microcontroller. Each oscillator pin is also separated from ground by a 20 pF nonpolar ceramic disc capacitor. Figure 3.13 shows the location of all the quartz crystal oscillators on the board. An electrical diagram for the oscillator setup is included in Appendix D.

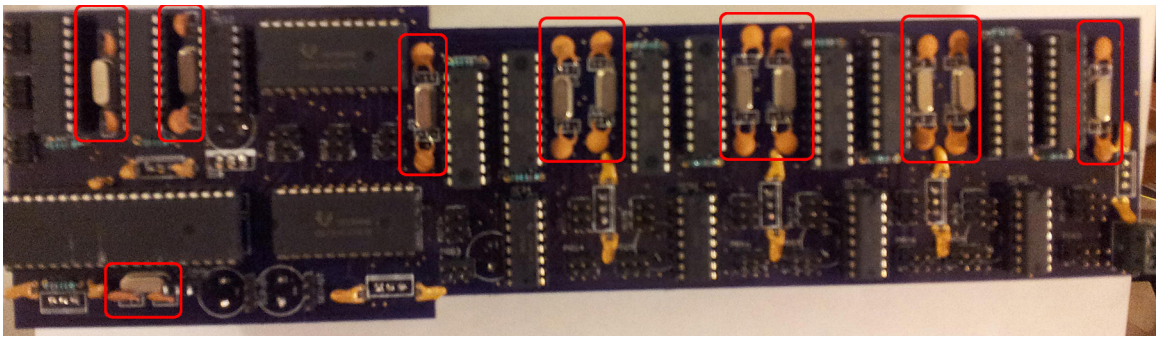


Figure 3.13: Location of all quartz crystal oscillators on the board

3.2.3 Master Microcontroller

The master microcontroller is an Atmel ATmega644p 40-pin chip. The ATmega644p was chosen over other chips because of its optimum selection of pin features and low cost. It is capable of two USART serial communication channels (both needed to communicate with both the computer and slave chips) as well as meeting all required input/output and timing requirements to interact with the slave chips and servo motors. The pinout is shown in Figure 3.14. Figure 3.15 shows the location of the master microcontroller on the board.

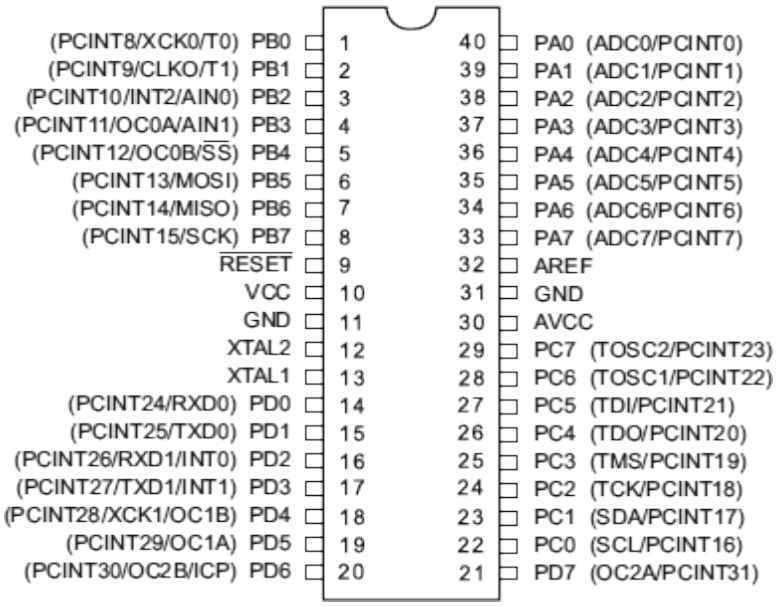


Figure 3.14: ATmega644p master microcontroller pinout

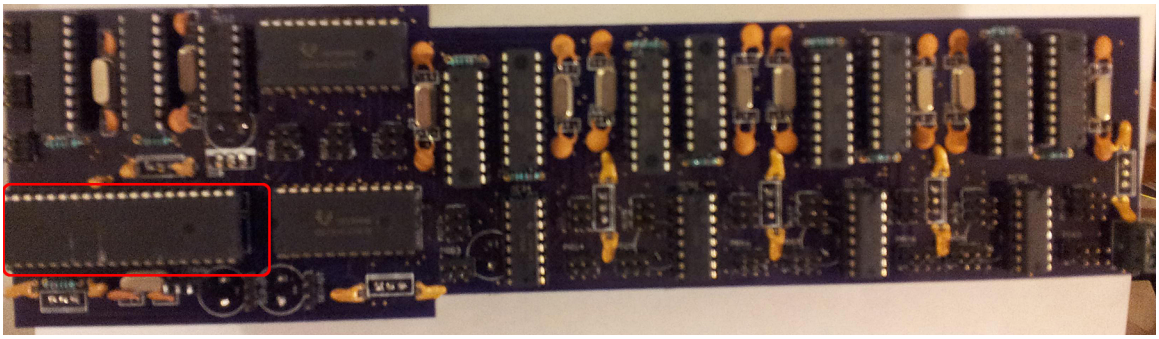


Figure 3.15: Location of the master microcontroller on the board

The main purpose of this microcontroller is to translate instructions from the computer and to output the correct desired positions to the slave motor controller chips. All major control calculations are outsourced to the slave chips to allow the master to focus almost entirely on communication, since the master must communicate with 10 slave chips and a computer. The master chip also controls both servo motors and the on/off index wag motion since those motions do not require repetitive calculations for accurate position control. Figure 3.16 shows the

location of the servo output ports on the board.

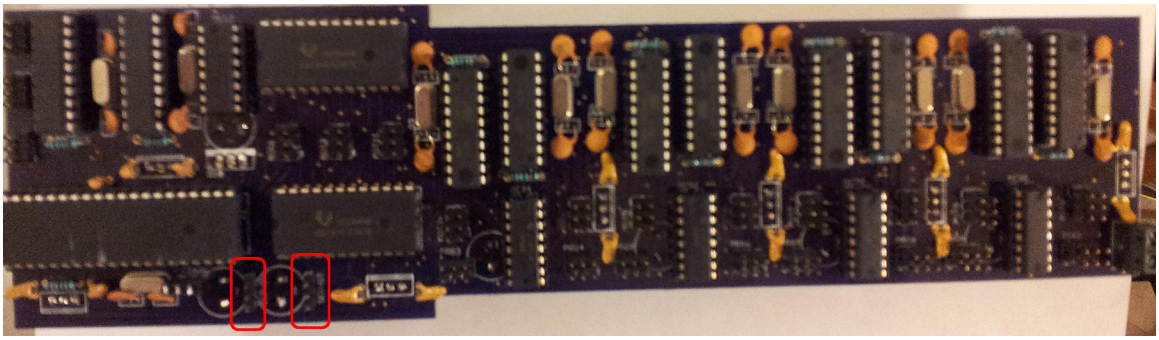


Figure 3.16: Location of the servo output ports on the board

3.2.4 Slave Motor Controllers

Each of the 10 slave chips is an Atmel ATtiny2313 20-pin microcontroller chip designed to output a PWM signal to the motor driver chips that control the output of the Maxon DC motors. The pinout for this chip is shown in Figure 3.17.

The 11th DC motor not controlled by a slave chip corresponds to the pull cable for the index-middle crossing motion. Since this is a simple on-off signal that doesn't require feedback, a separate slave chip is not required for control.

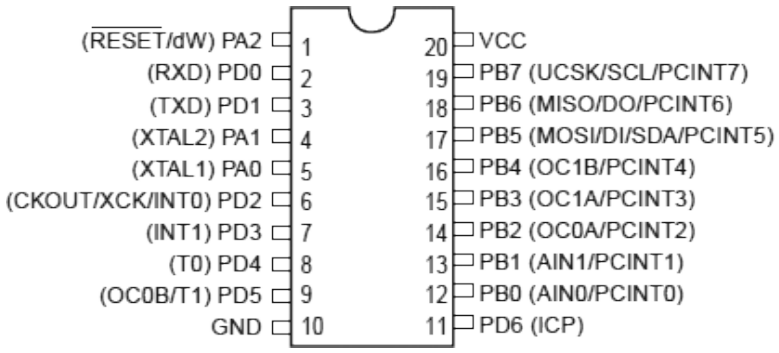


Figure 3.17: ATtiny2313 slave motor controller pinout

The slave chips that control the DC motors monitor the quadrature incremental encoders on the Maxon motors as they turn to keep track of the joint angles. Using

proportional control calculations, the slave chips calculate what voltage (in the form of a pulse-width modulation duty cycle) to send to the motors based proportionally on the difference between the current joint angle and the desired joint angle. Figure 3.18 shows the location of the slave motor controllers on the board.

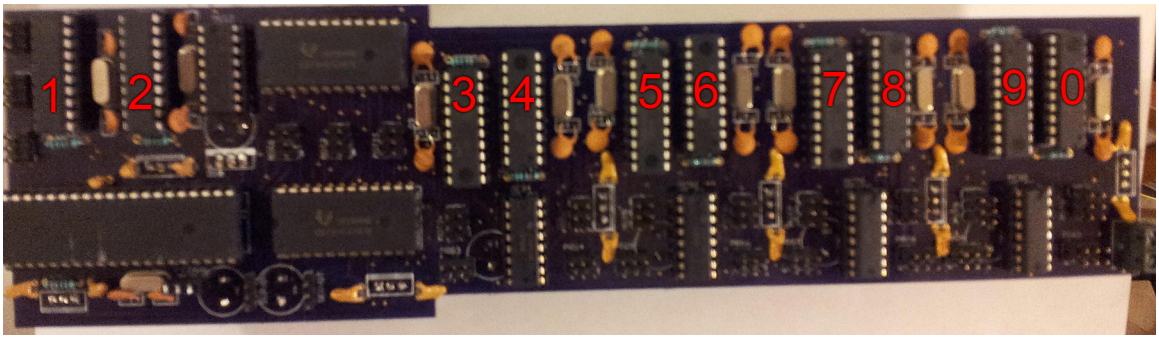


Figure 3.18: Location of the ATtiny2313 slave motor controllers on the board

3.2.5 Motor Drivers

To output an adequate power signal to the Maxon DC motors while isolating sensitive electrical components from the relatively large current required to drive the motors, 5 L293D quadruple half-h motor driver chips are used. The pinout for the L293D is shown in Figure 3.19. These chips can provide bidirectional control of 2 motors each (2 full H-bridges instead of 4 half H-bridges).

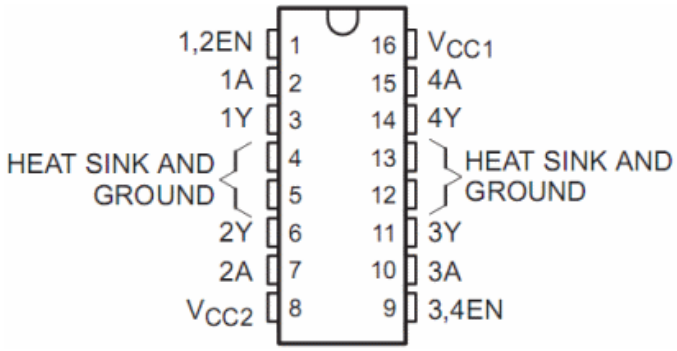


Figure 3.19: L293D motor driver pinout

Each H-bridge receives two directional inputs that determine the motor direction (or force the motor to brake) and a PWM duty cycle signal from the motor controller chips. The PWM signal switches the power output of the chip on and off to modulate the output voltage to the desired value between 0 and 7.2 V. Figure 3.20 shows the location of the motor drivers on the board. An electrical diagram for the controller-driver interactions is included in Appendix D.

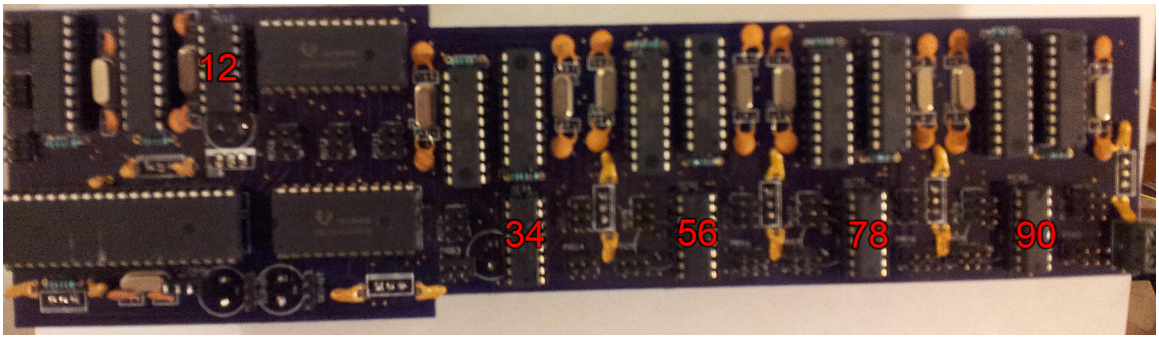


Figure 3.20: Location of the L293D motor drivers on the board

The single pull-cable motor is powered by a n-channel MOSFET activated by a single pin from the master microcontroller. Figure 3.21 shows the location of the MOSFET. This MOSFET comes in a standard TO-220 package for through-hole applications.

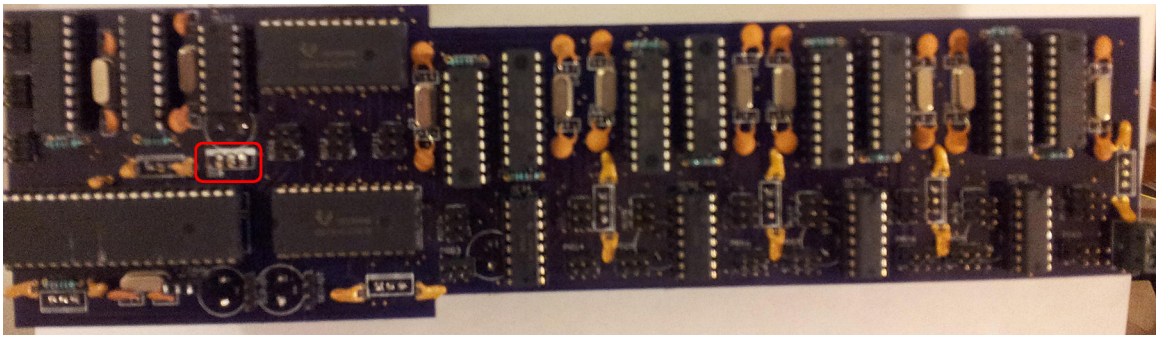


Figure 3.21: Location of the MOSFET controlled by the master microcontroller

3.2.6 Communication

The computer communicates with the master microcontroller via USB using the Sparkfun FT232R USB to serial breakout board. This device converts the incoming signals into USART RX (receive) and TX (transmit) signals for the microcontroller. A picture of the breakout board is shown in Figure 3.22.

Figure 3.22: Sparkfun USB to serial breakout board

The master microcontroller then communicates with its 10 slave chips over a second shared USART line. Both the RX and TX lines are connected to multiplexers to prevent all slave chips from receiving communication intended for only one chip. Routing on the multiplexers is controlled by the master microcontroller.

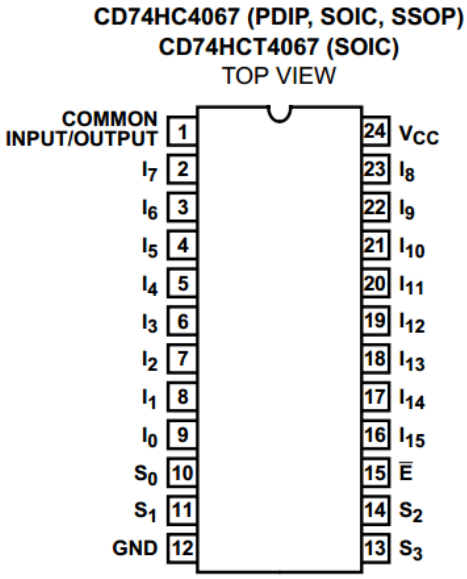


Figure 3.23: CD74HC4076E multiplexer pinout

The multiplexers chosen are CD74HC4076E through hole multiplexer chips made by Texas Instruments. These multiplexers can connect 16 output pins to a single input pin with 4 digital inputs from the master microcontroller used to select the output line. An electrical diagram for the communication setup is included in Appendix D. Figure 3.24 shows the location of the major communication components on the board.

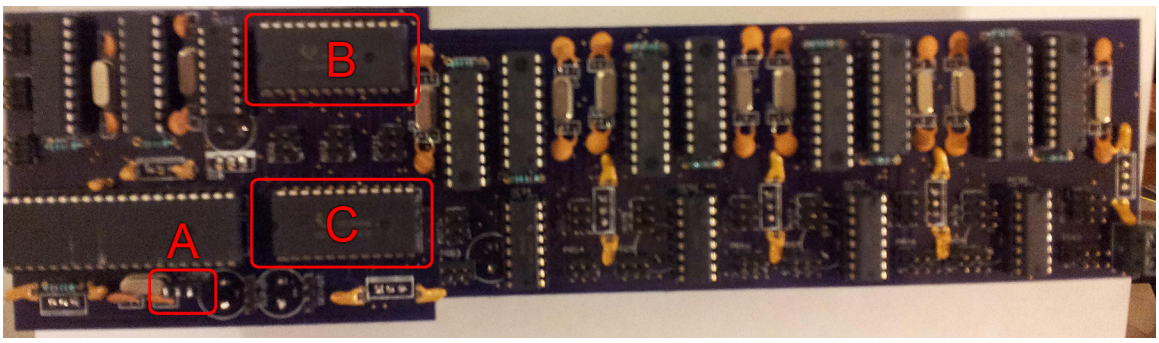


Figure 3.24: Communication components on the board. A) USB to serial breakout board output. B) Slave to master MUX. C) Master to Slave MUX.

3.2.7 Power

The robotic finger-spelling hand can be powered by any power source greater than 7 volts. Included with the hand are a 7.5V, 2.93A Mean Well AC adapter (5.5mm barrel jack plug) and a 7.2V NiMH RC car battery. These are shown in Figure 3.25. The hand does not have a charging circuit so the battery will not charge while the AC adapter is plugged in. A second battery and corresponding battery charger are included with the hand so the batteries can be charged and swapped if necessary. These power sources connect to the board using screw terminals at the bottom of the board.

Figure 3.25: AC adapter (left) and battery (right)

Power is stabilized using 1000uF capacitors and voltage regulators. These capacitors will draw energy from the power supply while the corresponding motors are stationary. This energy will be available for quick release when the motors turn. During battery operation, this will improve the life of the battery by allowing the capacitors to cope with spikes in power demand instead of forcing the battery to do so.

A 5V voltage regulator is used to stabilize the input power for every 2 slave motor controllers and their corresponding motor driver. Another 5V regulator stabilizes the input power for the master microcontroller and multiplexers. Power for the servo motors is stabilized by a 6V voltage regulator. All voltage regulators and MOSFETS come in TO-220 packages and are attached to compatible heat sinks

to prevent overheating. Figure 3.26 shows the location of all the major power components on the board.

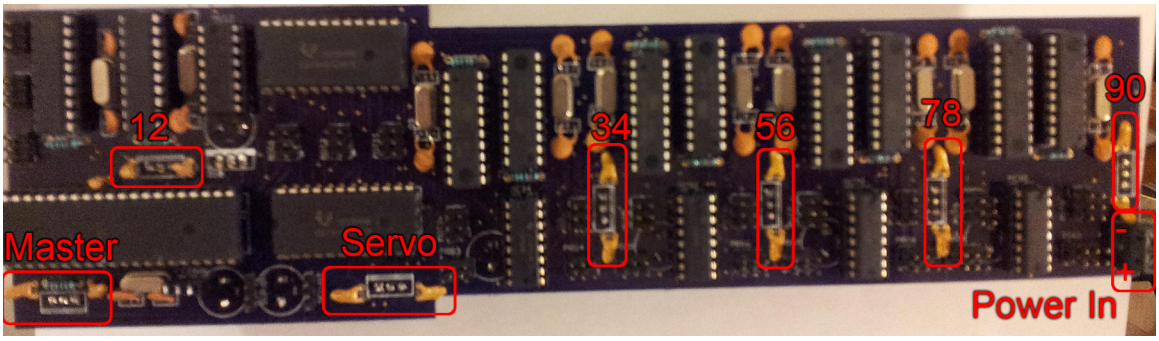


Figure 3.26: Location of power components on the board. 5V voltage regulators for microcontrollers, 6V voltage regulator for servos, and screw terminals for input power

An electrical diagram for the power setup is included in Appendix D.

Chapter 4: Programming and Software

4.1 Slave Motor Controller

Each slave motor controller has three main duties.

1. Interpret commands from the master microcontroller.
2. Monitor the motor head's position by reading the encoder channels.
3. Move the motor head to different positions depending on those instructions.

Tasks 1 and 3 are performed sequentially in the main code loop while task 2 is performed on demand whenever the encoder interrupt pins toggle. All the code for the slave motor controller chips can be found in Appendix G.

4.1.1 Communication and Data Task

All communication between the master and slave controllers consists of single-byte serial communication based on libraries simplified from the original code developed for Cal Poly's ME 405 Mechatronics course. These serial communication libraries were modified to fit on the small ATtiny2313 chips with unnecessary functions stripped out.

Upon startup, the slave controller code creates a serial port object based on the built-in USART protocol. The port is set to transmit and receive data at 9600 baud and is equipped with member functions to check if a character has been received, collect a received character from the character buffer, check if the transmitter is ready to transmit, and transmit a single byte to the master controller. The main loop communication code consists of the following:

1. Check if a character has been received. If not, skip the communication phase.
2. If a character has been received, collect and store it.

3. Interpret the stored character.

4. Execute a command depending on what character was received.

The communication sequence during the main loop occurs in a data handling task that parses incoming commands and reacts to them by manipulating corresponding command variables used by the motor output task. Table 4.1 shows a list of incoming commands and corresponding reactions.

Table 4.1: Slave Motor Controller Character Commands

Character	Corresponding Command
a,b,c,d, or e	Move motor to 0, 30, 45, 60, or 90 degrees respectively
S	Stop motor
G	Enable motor
C	Calibrate the encoders by clearing the encoder count
E	Send the encoder count to the master microcontroller
Any ASCII number	Load position and gain data for the corresponding motor number (0 = motor 10)

The process of command parsing requires the boolean check_for_char command ported from the ME405 USART libraries. Code Block 4.1 shows the waiting state of the communication/data task that only transitions to a new state once a character has been received. Once in the new state, the getchar command is used to retrieve the character. Since the getchar command waits until a character is actually received before exiting, this sequence of checking for a character before retrieving it prevents the microcontroller from getting stuck waiting in the retrieval process.

Code Block 4.1: Checking to see if a command has been received

```
if(sport.check_for_char())
    state_data = 1; // If character received go to state 1
else
    state_data = 0; // If not remain in state 0
```

Outputting a response to the master requires the send command, which sends only a single character. Code Block 4.2 shows the send command in action when

responding to a motor stop command from the master microcontroller.

Code Block 4.2: Example of send command usage

```
// S,G disable and enable the motor
case('S'): // Stop Motor
    flag_enable = false;  // Disable motor
    state_data = 1;       // Return to state 1 on the next loop
    sport.send('s');      // Confirm command reception
```

To keep code identical on all slave motor controllers, preprogrammed angular target position and gain data for all motors is stored on every chip. At any time the master microcontroller can reassign a particular slave a different motor number, though currently this only occurs during initialization. Once a slave motor is assigned a motor number, it loads position and gain data for that particular motor to use for position control calculations. Code Block 4.3 shows this process.

Code Block 4.3: Loading gain and pre-set angular position data

```
// Load angle data
for (i_angle = 0; i < 5; i++)
{
    set_point_angles[i_angle] = angles[i_angle][motor_number-1];
}

// Load gain data
kp = kp_array[motor_number-1];
```

4.1.2 Encoder Reading

Due to the speed that each motor turns, the quadrature encoders channels must be read through interrupt pins. This allows the microcontroller to respond instantly to pin changes without waiting for other code to finish processing to ensure that the encoder count is as accurate as possible. Both interrupt service routines are identical and consist of the following code structure:

1. Store last reading to a new variable.

2. Read both encoder pins, concatenate, and store as a 2 digit binary number.

3. Determine direction based on current reading and previous reading.

4. Increment or decrement count accordingly.

Code Block 4.4 shows the code used in the interrupt service routine to achieve this code structure.

Code Block 4.4: Interrupt service routine for encoder reading

```
// Interrupt for Encoder Channel A
ISR(INT0_vect)
{
    previous_reading = current_reading; // Store last reading to old
        variable

    // Take in new reading (A << 1) | B
    current_reading = ((PIND & 0b00000100) >> 1) | ((PIND & 0b00001000)
        >> 3);

    // Evaluate Reading
    switch(current_reading)
    {
        case(0):              // Current value == 00
            switch(previous_reading)
            {
                case(1):      // 01 to 00
                    CLOCKWISE ; // (count++)
                    break;
                case(2):      // 10 to 00
                    COUNTERCLOCKWISE ; // (count--)
                    break;
                default:      // 11 to 00
                    ENCODER_ERROR ;
                    break;
            }
            break;
        ...
        // more cases for current value == 01, 10, or 11
    }
}
```

Encoder quadrature uses the 90 degree phase misalignment of the two encoders to indicate the direction of motion. When moving left to right in the sequence 00 01 11 10 00, the motor is moving back towards the start position (zero). When moving right to left in the same sequence, the motor is turning away from the start position.

See Figure 4.1 for a visual representation.

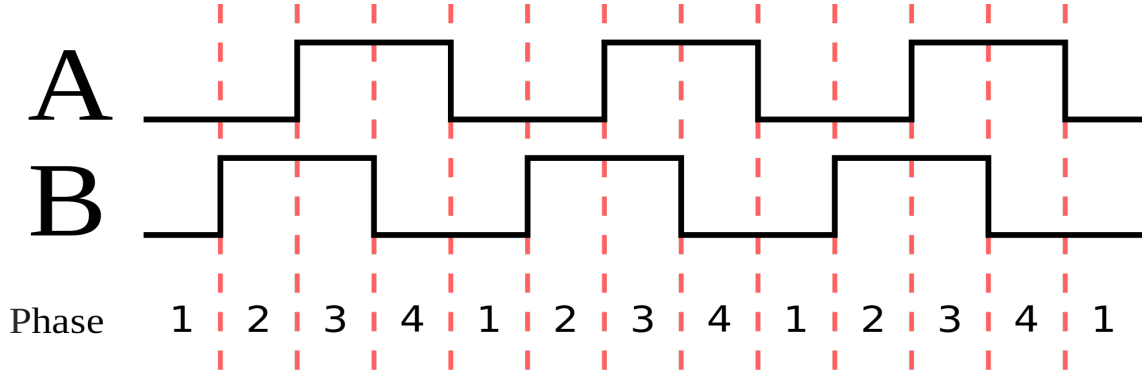


Figure 4.1: Encoder quadrature system for two incremental encoder channels.

The beginning of the main loop also contains code to trim the encoder count within acceptable bounds before performing any motor control calculations. Though the physical constraints of the actual hand do not allow movement beyond roughly 90 degrees, this safety measure prevents accidental overflows from causing jarring movements in the fingers due to sudden changes in the encoder count.

4.1.3 Motor Output Task

The slave motor controllers are designed to move the motor to pre-set angular positions roughly close to 0, 30, 45, 60, and 90 degrees. These positions are preprogrammed into the code for all motor controllers. This simplifies communication between the master and slave microcontrollers while still being able to represent all of the necessary finger joint angles required for fingerspelling.

The calculation of the input signal occurs during every pass of the controller chip's main loop. Once the main loop is done with communication (character command executed or no input character detected) motor output calculation commences. That process consists of the following steps:

1. Determine whether the current encoder count is higher or lower than the

desired count.

2. Set the motor direction and calculate the control error accordingly.

3. If the difference exceeds the control error tolerance, calculate a motor output PWM duty cycle proportional to the magnitude of the control error.

4. Trim the output PWM value if it exceeds the maximum duty cycle (255) and save it as an 8 bit unsigned character.

5. Check if the motor on/off flag is on. If so, set the calculated duty cycle. If not, stop the motor.

Steps 1-4 are shown in Code Block 4.5.

Code Block 4.5: Motor output value calculations

```
// Calculate control loop error
control_error = count - desired_count;

// Calculate value and trim to 0-255 range
if (control_error > 1)
    motor_output = (long) (kp * control_error * 255) / 768;
else if (control_error < -1)
    motor_output = (long) (0xFF * control_error * 255) / 768;
else
    motor_output = 0;
if (motor_output > 255)
    motor_output = 255;
```

The actual output process calls upon the member functions of the motor object class. These functions can stop the motor, set the motor direction, or change the voltage output to the motor (8 bit value). The actual output code from the motor task is shown in Code Block 4.6.

Code Block 4.6: Motor output commands

```
// Set direction
if (control_error > 0)
{
    mtr.d1();  // Set the motor to output direction 1
```

```
    mtr.output( (unsigned char) motor_output); // set output value
}
else if (control_error < 0)
{
    mtr.d0();  // Set the motor to output direction 0
    mtr.output( (unsigned char) motor_output); // set output value
}
else
    mtr.stop(); // Brake
```

4.2 Master Microcontroller

The master microcontroller performs two main tasks. The user interface task interacts with the user to retrieve commands. The output task translates commanded outputs from the user interface task into serial port commands for each motor.

The master microcontroller has several duties:

1. Interpret commands from the computer
2. Display a simple user interface for the user
3. Parse sentence inputs from the user to send commands to the slave motor controllers
4. Control the MOSFET that handles the pull cable motor
5. Send a command signal to the servo motors that control the wrist

4.2.1 Communication

The master microcontroller has two USART communication lines, one communicating with the entire set of slave motor controllers (RX1/TX1), and the other communicating with the a computer serial port terminal (RX0/TX0). These lines are set up to communicate at 9600 baud with 8 data bits, 1 stop bit, and no parity (9600,N,8,1). This should be the default in most serial terminal programs.

The software for the USART protocols is taken directly from the serial communication code libraries used in Cal Poly's ME 405 Mechatronics course. These libraries and included methods are capable of performing the following functions:

- Check whether a character has been received by the microcontroller
- Check whether the serial port is ready to send a character
- Retrieve a character that has been received by the microcontroller
- Send a character through the serial port to another microcontroller or to the connected computer
- Convert a character to a specified displayable format when being output to the computer

4.2.2 User Interface Task

The user interface is a command line interface that takes inputs from the keyboard and translates them to commands to output to the motors. It is one of the two main tasks in the master chip and runs every 25 milliseconds.

Main Menu

Upon startup, the user is given several options in the main menu:

Table 4.2: Main Menu

Character	Corresponding Command
ESC	Stop all motors
C	Calibrate an individual motor
ENT	Enter a sentence for the hand to spell
E	Retrieve the current encoder count from the encoder
M	Manual mode (send any single-character command to a chosen motor)

The stop all motors command (ESC) sends the stop command described in Table 4.1 (S) to each individual motor. The calibrate command (C) presents the

user with a list of motors and clears the encoder for the chosen motor by sending the calibrate command described in Table 4.1 (C) to it. The enter sentence command (ENT) prompts the user to enter a sentence for the hand to translate to fingerspelling (described more below). The retrieve encoder count and manual mode commands are intended for diagnostic and setup purposes to compensate for the fact that the user interface does not communicate directly with the slave motor controllers.

Sentence Parsing

When a user enters a sentence into the sentence prompt, each individual letter is stored in a character queue (maximum sentence length is 255 characters). The code libraries for this character queue were taken directly from the ME 405 libraries. Each character in the queue is analyzed to determine whether it is a character that can be fingerspelled (alphanumeric). If it is, then it is submitted to the output task by calling one of the output task's methods.

If the character is a punctuation character, it is ignored unless it is a pause character (comma, period, or space). Once the last character has been processed, the program will prompt for another sentence input.

Spellable characters are translated into finger gestures in the output task. Pause characters extend the delay before another set of motor commands is sent to the hand.

4.2.3 Output Task

The output task consists of the code required to translate a character into the proper outputs required to command the motors to move to different positions. The output task gets called by the task scheduler every 10 milliseconds.

Hand Configurations

Each finger has a limited number of possible configurations used in fingerspelling. Each finger configuration has a corresponding function that contains all of the necessary motor commands required to achieve that configuration at every joint in the finger. The index, middle, ring, and pinky fingers share three configurations in common:

- **Stretched:** Extended straight up
- **Curled:** Curled as if gripping a ball
- **Clenched:** Clenched as if making a fist

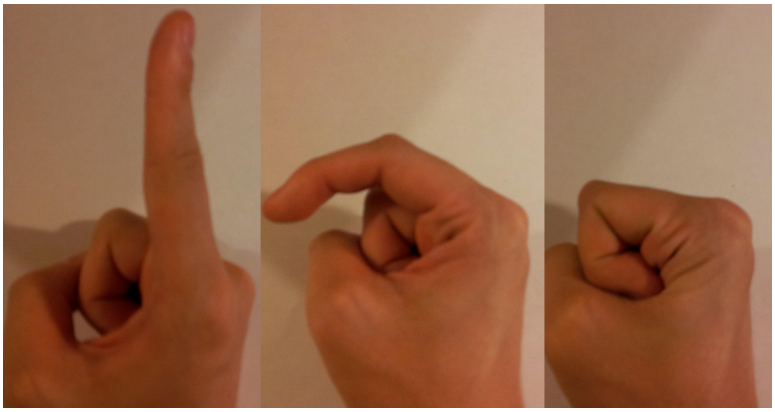


Figure 4.2: Stretch, curl, and clenched configurations common to all fingers

Figure 4.2 shows these three configurations. The index and middle finger also have extra configurations since these fingers have more than one degree of freedom. Two configurations shared by these two fingers are the vertical clench and fold configurations shown in Figure 4.3.

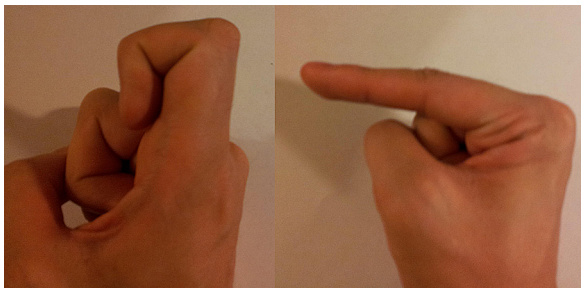


Figure 4.3: Vertical clench and fold configurations used by the index and middle fingers

The pull-cable motor on the index finger requires two special configurations for the wagging motion to form the letters U and R. The U configuration simply pulls the stretched index finger next to the stretched middle finger to close the gap between them. The R (cross) configuration does the same but bends the proximal metacarpal joint 45 degrees to allow the index finger to cross under the middle finger. These configurations, along with a normal configuration, are shown in Figure 4.4.

Figure 4.4: Normal, U, and cross configurations used only on the index finger

The thumb also has many configurations due to having four degrees of freedom. These configurations are

- **Flat up:** Flat up against the side of the index finger
- **Folded up:** Same as flat up but folded on top of the palm
- **Folded in:** Folded over the palm

- **Folded out:** Same as folded in but forming a 90 degree angle with the palm
- **Stretched:** Extended out away from the palm
- **Curled:** Same as folded out but with the distal portion of the thumb curled up

Figure 4.5: Thumb configurations. Flat up, folded up, folded in, folded out, stretched, curled

Each possible output character consists of a collection of thumb, finger, and wrist configurations. When the output task receives a new character from the user interface task, the character is filtered through a list of configurations for every spellable character. The list of configurations for every character is located in Appendix B.

If a character's collection of configurations can cause potential collisions when forming future characters (such as the thumb wrapped over the fingers when forming the letter S), the fingers that can cause these interferences are identified in the character definition. Before each character is formed, fingers that have been flagged as interfering are "opened" (pulled away from the palm) before the rest of the character is formed. Characters that may result in finger interference during formation are formed in multiple steps with the interfering fingers moved last.

Motor Output

The output process for each of the motors consists of sending the single-character commands “a”, “b”, “c”, “d”, or “e” through the serial port to the chosen motor as code for finger joint angles of 0, 30, 45, 60, and 90 degrees respectively. The actual encoder counts required to achieve these positions can be tuned in the code for the individual slave motor controllers. Code Block 4.7 shows an example of the function calls used to fully extend the index finger (0 degrees to both joints).

Code Block 4.7: Master motor output example

```
void task_output::index_stretch(void)
{
    output_to_motor(1,'a');
    output_to_motor(9,'a');
}
```

Pull-Cable Motor Output

The single pull-cable motor required to form the letters U and V operates in an on/off manner and therefore does not require position control. To control the MOSFET that powers that motor a general purpose input/output pin is connected to the gate of the MOSFET. When the pin is toggled on, the MOSFET allows current through and turns the motor on. When the pin is toggled off, no current is allowed through and the spring on the finger joint restores it to its natural state. Code Block 4.8 shows an example of this using the index_cross index finger configuration. Sending the 1 to motor 11 using the same output_to_motor command as with normal motor outputs activates the pull-cable motor. Sending 0 to motor 11 turns the motor off.

Code Block 4.8: Pull-cable motor example

```
void task_output::index_cross(void)
{
```

```
    output_to_motor(1,'c');
    output_to_motor(9,'a');
    output_to_motor(11,1);
}
```

Servo Control

Hobby servo motors use a special pulse-width signal to communicate the desired angular position. This signal is a square wave with 20 millisecond period. The pulse width varies from 1 ms to 2 ms with 1 ms representing 0 degrees and 2 ms representing 180 degrees. All pulse widths in between scale proportionally with angles between 0 and 180 degrees. See Figure 4.6 for a visual representation of this signal format.

Figure 4.6: Servo PWM signal format

To generate this signal, the 16-bit output-compare feature used for PWM outputs is used. When a desired angle is called for, a 16-bit value that represents the desired pulse-width of the square wave (0 min, 65535 max) is calculated and stored in the output compare register. When the corresponding 16-bit timer/counter has a value between 0 and the output compare value, the corresponding output pin outputs 5V. When the timer/counter has a value between the output compare value and 65535, the output pin outputs 0V. The timer/counter resets at 0 when it overflows its 16-bit register.

Code Block 4.9 shows the process of changing the desired servo angles for both servos. Sending an angle between 0 and 180 degrees to motor 12 or 13 using the same output_to_motor command as with normal motor outputs causes the

corresponding servo object to recalculate the output compare register value based on the new desired angular position.

Code Block 4.9: Servo control example

```
void task_output::wrist_bent(void)
{
    output_to_motor(12,90);
    output_to_motor(13,0);
}
```

Chapter 5: Testing

The following tests and evaluations have been performed on the hand, the circuits, and the software with the following results:

5.1 Current Project Status

5.1.1 Hardware

The mechanical design is complete and has been fully fabricated. When powered separately from the electrical hardware, the motors actuate the fingers as designed. The electrical hardware design is also complete and has also been fully fabricated.

5.1.2 Software

Development has stalled because the ATtiny2313 slave motor controllers keep resetting themselves. This problem was discovered when a command to print a character to the screen via the serial port ran repeatedly despite the fact that it was located outside the main loop.

This problem still occurs when an individual chip is tested on a breadboard separate from the printed circuit board used in the project. This problem also does not occur on the master microcontroller.

Normal occurrences of this phenomenon are a result of a failure to pull the /RESET pin high using a pull-up resistor. A breadboard test and oscilloscope monitoring on the main board indicate that this is not the cause of this particular problem.

Attempts to check the MCUCSR register that normally provides some indication of the cause of the latest reset result in none of the MCUCSR bits being high. This might suggest that the problem is strictly software-related. Attempts to run a completely stripped version of the program code also do not result in this

reset behavior. This also points to a software-related problem. Further debugging still needs to be conducted to narrow down the source of the problem, but something in the code is causing a jump back to the very beginning of the code.

5.2 Evaluation of Design Specifications

Certain specifications have been evaluated if they did not require completion of development to be tested. The full list of specifications can be found in Table 2.2

5.2.1 Dimensional Specifications

Specifications requiring the hand to remain within a certain size and shape have been met since they were repeatedly evaluated throughout the design process. These specifications are maximum palm thickness (30 mm) and maximum deviation from average limb size (25 %). The thickness of the palm is roughly 29 mm, and aside from a slightly enlarged palm, all of the dimensions used to rapid-prototype the hand were taken from an actual human hand.

5.2.2 Weight

The weight specification required the hand and all internal parts to weigh less than 10 lbs for safety and portability. The entire hand weighs 4.3 lbs, well below the 10 lb limit. When fully assembled, the hand can easily be transported.

5.2.3 Assembly and Disassembly

The specifications covering difficultly of assembly and disassembly are assembly time and disassembly time (both 20 minutes) for the prototype. The latest design fails to meet all of these requirements.

Assembly requires at least 6 hours of careful cable threading and tightening for the mechanical portion. Disassembly requires a similar amount of time to remove the metal rods connecting each joint and remove all the previous cabling. With practice, a technician may be able to cut this time by a few hours, but will still fall well short of the 20 minute assembly/disassembly time goals.

The latest design is significantly more difficult to assemble, disassemble, and repair than initially envisioned.

5.2.4 Specifications Not Tested

Specifications relating to the operation of the hand have not been tested since the hand does not currently work. These specifications are accuracy, letter formation rate, number of broken parts, and number of pinches, burns, and shocks as a result of operation.

Chapter 6: Conclusions

6.1 Conclusions

Though this particular iteration of the project has not been completed, the project was still a worthwhile endeavor. This specific version of the project serves as an experimental example that explored many different potential solutions and discovered that some of them failed. This information is valuable for future projects as a guide for what pitfalls to avoid, which will save future research teams time and money. Suggestions from this project have already been implemented successfully in other iterations of this project.

The feasibility of a robotic hand hardware capable of fingerspelling has been proven all the way up to the division between hardware and software. This hand does work mechanically and will work under electrical control when the software is fixed.

6.2 Recommendations

Several targeted improvements can be made to the hand design and to the project itself to make it more reliable both mechanically and electrically. Many of these suggestions are also geared towards removing major frustrations that resulted from the current design.

6.2.1 Mechanical Recommendations

Many of the problems encountered are a result of using string/cables/wire to transfer motion from the motors to the joints for bidirectional control.

- Custom pulleys had to be rapid prototyped because of too much slippage between the cables and pulleys.

- Finding a cable that was both flexible and strong was difficult. Dental floss frayed too easily. Dental tape slipped or snapped too easily. Sewing thread snapped very easily. Eventually fishing line was chosen.

- Keeping the fishing line in tension was difficult in such a small area. Having so many lines in one place crammed with all the motors and routing posts made using springs to keep the lines in tension impossible. The only springs that could fit in such a small area were not stiff enough.

- Attaching fishing line to the joint pulleys and motor pulleys was challenging as well. The line had to be wrapped around each pulley multiple times and often jumped out of the pulleys if they were loosened for just a few seconds.

- The bidirectional fishing line loop connecting the motor and joint provides little to no resistance to external forces. Only the software position control keeps the finger joints at the correct angles when forces are applied to the fingers.

- Since knots were too difficult to tie while keeping every line in tension, the best method for fastening the fishing line was to crimp the line. This method is permanent and the crimp (and sometimes the fishing line itself) must be destroyed whenever the motor or finger joint needs to be re-strung.

Choosing to not use cabling altogether might have prevented some of these problems from arising. Future versions of the hand should explore the use of gears as a means for motion transmission. However, even while using cabling, several other design changes could have made using cables less of a challenge in hindsight.

- Moving the actuators out of the palm and into the wrist/forearm area would have allowed enough cable length and space to use stiffer springs for

tensioning. This also removes weight from the palm resulting in less torque needed to move the wrist joints.

- Using the actuators to only pull one cable in one direction and relying on a return rotational spring to restore the joint to a neutral position when the actuator lets off drastically simplifies the setup and provides much stiffer resistance to external forces on the fingers.

Stronger actuators are also needed for better performance. Motors with more torque would more easily overcome the friction in some of the joints, especially in the ones closest to the fingertips. Moving actuators out of the palm and into the wrist, as suggested before, will also allow larger actuators to be chosen with space being less of a problem.

The combination of a DC motor, gearhead, and encoder is not the best design choice for this project in terms of complexity. Most of the circruitry in the main board is required to essentially build a servomechanism, something that is already pre-packaged and ready to use in hobby servos. Development boards like the Arduino already have servo libraries. The code can be drastically simplified when not having to program and tune 10 different motor controllers.

Since robotic hands in general are so mechanically complex, better care should be taken during the design process to make sure that the assembly process is easier to complete. The combination of small parts and tight spaces can be very frustrating during assembly and repair.

When designing part sizes and shapes, too much focus was placed on weight reduction. This resulted in some parts being too fragile while the overall weight of the hand came in well under the maximum allowed weight defined in the specifications. Future design efforts should focus on designing solid, reliable hand parts first to avoid problems with stress concentrations and breaking parts. Once

the solid makeup of the hand is finished, the process of weight reduction by hollowing out parts can begin.

6.2.2 Electrical Recommendations

Several ideas for electrical components were dropped from this project to save time and reduce the project's complexity. Future teams working on building new hands should attempt to reincorporate these ideas back into the design.

- Bluetooth communication is another communication protocol that is compatible with the USART functionality of the Atmel microcontrollers. Setting up Sparkfun's Bluesmirf Bluetooth module is similar to using the FT232R breakout board for USB connectivity. The challenge will come from trying to incorporate both into the same circuit.

- A battery charging circuit similar to a laptop setup was initially proposed for the hand. The charger should be able to recharge the battery when plugged into an AC outlet while still having enough power to operate the hand. This would prevent the user from having to swap drained batteries and would also remove the need to purchase a separate battery charger.

- This project neglects a dedicated graphical user interface. Instead it relies on the master microcontroller printing characters over the serial port to the screen to interface with the sighted user. While this setup is simple and fast, text-only interfaces are not the best designs from a usability standpoint.

Since the start of this project, advances have been made to existing development boards, like the Arduino, and new development boards have been released, most notably the Raspberry Pi. Future project teams should re-examine the use of these boards, since the custom circuit board used in this project cost more to print than the purchase price of the more popular development boards.

6.2.3 Overall Project Structure and Scope

Most versions of this project have tackled both the mechanical and electrical phases of designing a robotic finger-spelling hand. At this point splitting the project into two separate projects, one mechanically oriented and one focused on the electronics and programming, will allow each team to develop more robust designs without having to juggle problems from multiple phases of the project.

While the teams can still communicate, especially since both teams must deal with the actuators, separate teams would allow more concentration on smaller phases of the project like fixing the power transmission system, implementing a battery charging circuit, or programming a GUI for the computer.

BIBLIOGRAPHY

[1] Barbara Miles. Overview on deaf-blindness. Technical report, National Consortium on Deaf-Blindness, 2008.

[2] National Association of Regulatory Utility Commissioners Board of Directors. *Resolution to Support Equal Access to Communication Technologies by People with Disabilities in the 21st Century*, February 2008.

[3] Mark Schalock and Robbin Bull. The 2010 national child count of children and youth who are deaf-blind. Technical report, National Consortium on Deaf-Blindness, 2011.

[4] Andrew Y.J. Szeto and Kathee M. Christensen. Technological devices for deaf-blind children: Needs and potential impact. *IEEE Engineering in Medicine and Biology Magazine.*, pages 25–29, September 1988.

[5] Barbara Miles. Talking the language of the hands to the hands. Technical report, National Consortium on Deaf-Blindness, October 2003.

[6] David L Jaffe. Evolution of mechanical fingerspelling hands for people who are deaf-blind. *Journal of Rehabilitation Research and Development.*, 31(3):236–244, August 1994.

[7] Bill Vickars. American sign language fingerspelling & numbers: Introduction. Web Site. http://www.lifeprint.com/asl101/fingerspelling/fingerspelling.htm.

[8] Laurie Walcott. Interview with Laurie Walcott, Cal Poly Disability Resource Center, March 2010.

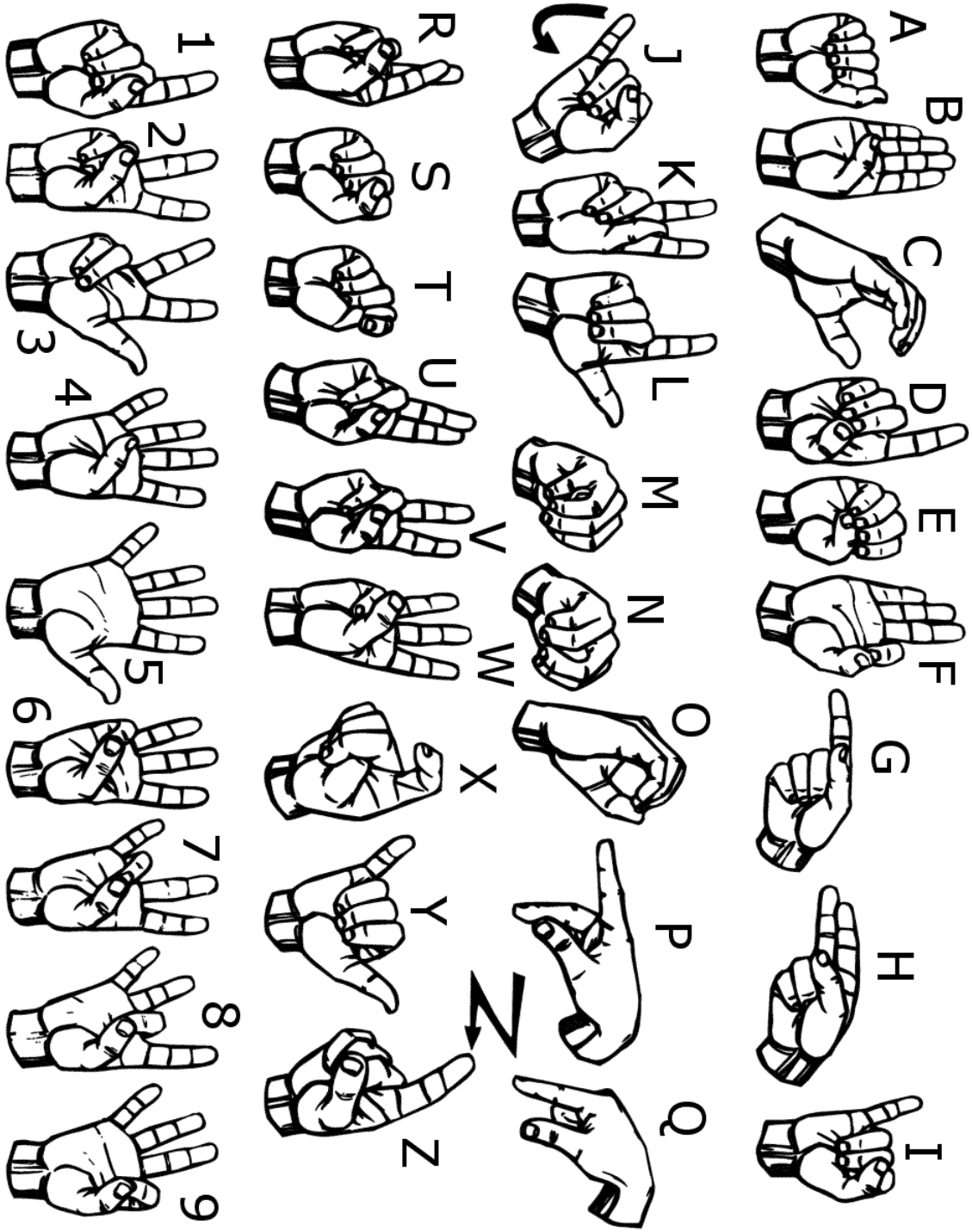
[9] Humanware. Deafblind communicator. Product Brochure. http://www.humanware.com/en-usa/products/deafblind_communication_solutions/deafblind_communicator.

[10] Audrey Steever, Luke Torgesen, and Chun Kau Cheung. Robotic finger spelling hand design report. Mechanical engineering senior project., California Polytechnic State University., 2008.

[11] Mario Garcia. Robotic finger-spelling hand for people with deaf-blind disabilities. Incomplete draft report submitted to Smith-Kettlewell Eye Research Institute.

[12] Henry Gray. *Anatomy of the Human Body.* Lea & Febiger, Philadelphia, PA, 1918. Public Domain edition from Bartleby.com, 2000. http://www.bartleby.com/107/.

Appendix A: American Manual Alphabet



Appendix B: Finger Configurations

Table B.1: Table of Finger Configurations

Char	Step	Thumb	Index	Middle	Ring	Pinky	Wrist
0,O	1	curl	curl	curl	curl	curl	default
1	1	flat up	stretch	clench	clench	clench	default
2,V	1	flat up	stretch	stretch	clench	clench	default
3	1	stretch	stretch	stretch	clench	clench	default
4,B	1	fold out	stretch	stretch	stretch	stretch	default
	2	fold in*					
5	1	stretch	stretch	stretch	stretch	stretch	default
6,W	1	fold out	stretch	stretch	stretch	clench	default
	2	fold in*					
7	1	fold out	stretch	stretch	clench	stretch	default
	2	fold in*					
8	1	fold out	stretch	clench	stretch	stretch	default
	2	fold in*					
9	1	flat up	clench	stretch	stretch	stretch	default
A	1	flat up	clench	clench	clench	clench	default

*Causes interference

Char	Step	Thumb	Index	Middle	Ring	Pinky	Wrist
B,4	1	fold out	stretch	stretch	stretch	stretch	default
	2	fold in*					
C	1	fold out	curl	curl	curl	clench	default
D	1	curl	stretch	curl	curl	clench	default
E	1	fold out	stretch	stretch	stretch	stretch	default
	2	fold in*	curl*	curl*	curl*	curl*	
F	1	flat up	clench	stretch	stretch	stretch	default
G	1	flat up	stretch	clench	clench	clench	bent
H	1	flat up	stretch	stretch	clench	clench	bent
I	1	flat up	clench	clench	clench	stretch	default
J	1	flat up	clench	clench	clench	stretch	default
	2						bent
	3						bent & twisted
	4						twisted
K	1	flat up	stretch	stretch	clench	clench	default
	2	fold up*					
L	1	stretch	stretch	clench	clench	clench	default
M	1	fold in*	stretch	stretch	stretch	clench	default
	2		v.clench*	v.clench*	curl*		

*Causes interference

Char	Step	Thumb	Index	Middle	Ring	Pinky	Wrist
N	1	fold in*	stretch	stretch	clench	clench	default
	2		v.clench*	v.clench*			
0,0	1	curl	curl	curl	curl	curl	default
P	1	fold up*	stretch	fold*	clench	clench	bent
Q	1	fold out	fold	clench	clench	clench	bent
R	1	flat up	cross*	clench	clench	clench	default
S	1	fold out	clench	clench	clench	clench	default
	2	fold in*					
T	1	flat up	v.clench*	clench	clench	clench	default
	2	fold in*					
U	1	flat up	stretch	stretch	clench	clench	default
	2		U*				
V,2	1	flat up	stretch	stretch	clench	clench	default
W,6	1	fold out	stretch	stretch	stretch	clench	default
	2	fold in*					
X	1	fold out	stretch	clench	clench	clench	default
	2	fold in*	v.clench				
Y	1	stretch	clench	clench	clench	stretch	default
Z	1	flat up	clench	clench	clench	stretch	Z1
	2						Z2
	3						Z3
	4						Z4

Appendix C: Parts List

Table C.1: Table of Electrical Parts

Part	Quan.	Part #
10 kΩ TE Connectivity Metal Film Resistor	11	LR1F10K
0.1 μF Vishay Tantalum Capacitors	7	199D104X9035A1V1E3
0.33 μF Vishay Tantalum Capacitors	7	199D334X9035A1V1E3
5V Diodes Inc LDO Voltage Regulator	6	AP1117T50L-U
6V ON Voltage Regulator	1	MC7806ACTG
20 MHz Vishay Crystal Oscillators	11	XT9S20ANA20M
20pF Xicon Ceramic Disc Capacitors	22	140-50N5-200J-RC
International Rectifier n-MOSFET	1	IRLB8721PBF
Aavid Thermalloy Heat Sinks	8	577202B00000G
#4-40 1/4" Screws	8	
#4-40 Nuts	8	
1000 μ Kemet Aluminum Electrolytic Capacitor	7	ESH108M010AH1AA
TI Multiplexer	2	CD74HC4067E
40-pin Atmel Microcontroller	1	ATMEGA644P
20-pin Atmel Microcontroller	10	ATTINY2313
16-pin TI Motor Driver	5	L293DNE

Part	Quan.	Part #
TE Connectivity 40-pin DIP Socket	1	1-390262-5
TE Connectivity 20-pin DIP Socket	10	1-390261-6
TE Connectivity 16-pin DIP Socket	5	1-390261-4
Duratrax DTX 4600 NiMH Battery	1	DTXC2149
Mean-Well AC Adapter 22W 7.5V 2.93A	1	GS25U07-P1J
Sparkfun FT232RL USB to Serial Breakout Board	1	BOB-00718

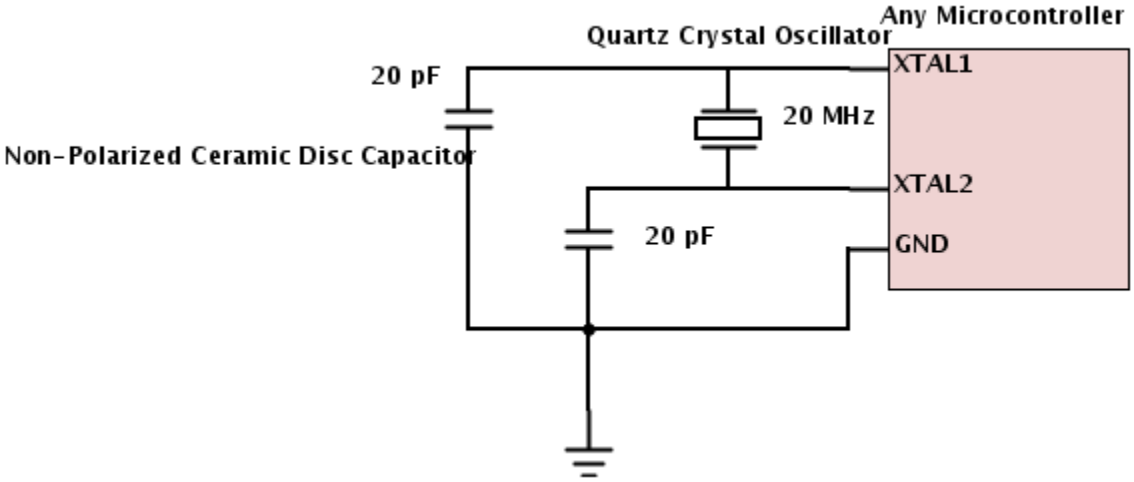
Table C.2: Table of Mechanical Parts

Part	Quan.	Part #
Maxon Motors	11	256105, 218416, 138061
Hitec HS 255 BB Hobby Servo	1	31225S
Hitec HS 985 MG Hobby Servo	1	32985S
Custom Motor Pulleys	10	Rapid-Prototyped
2.5” PVC Pipe with Bell	1	
#6-32 Screws	10	
1/16” Music Wire		
Thumb Distal Phalanx	1	Rapid-Prototyped
Thumb Proximal Phalanx	1	Rapid-Prototyped
Thumb MetacaRapid-Prototypedal Phalanx	1	Rapid-Prototyped
Thumb Fold Knuckle	1	Rapid-Prototyped

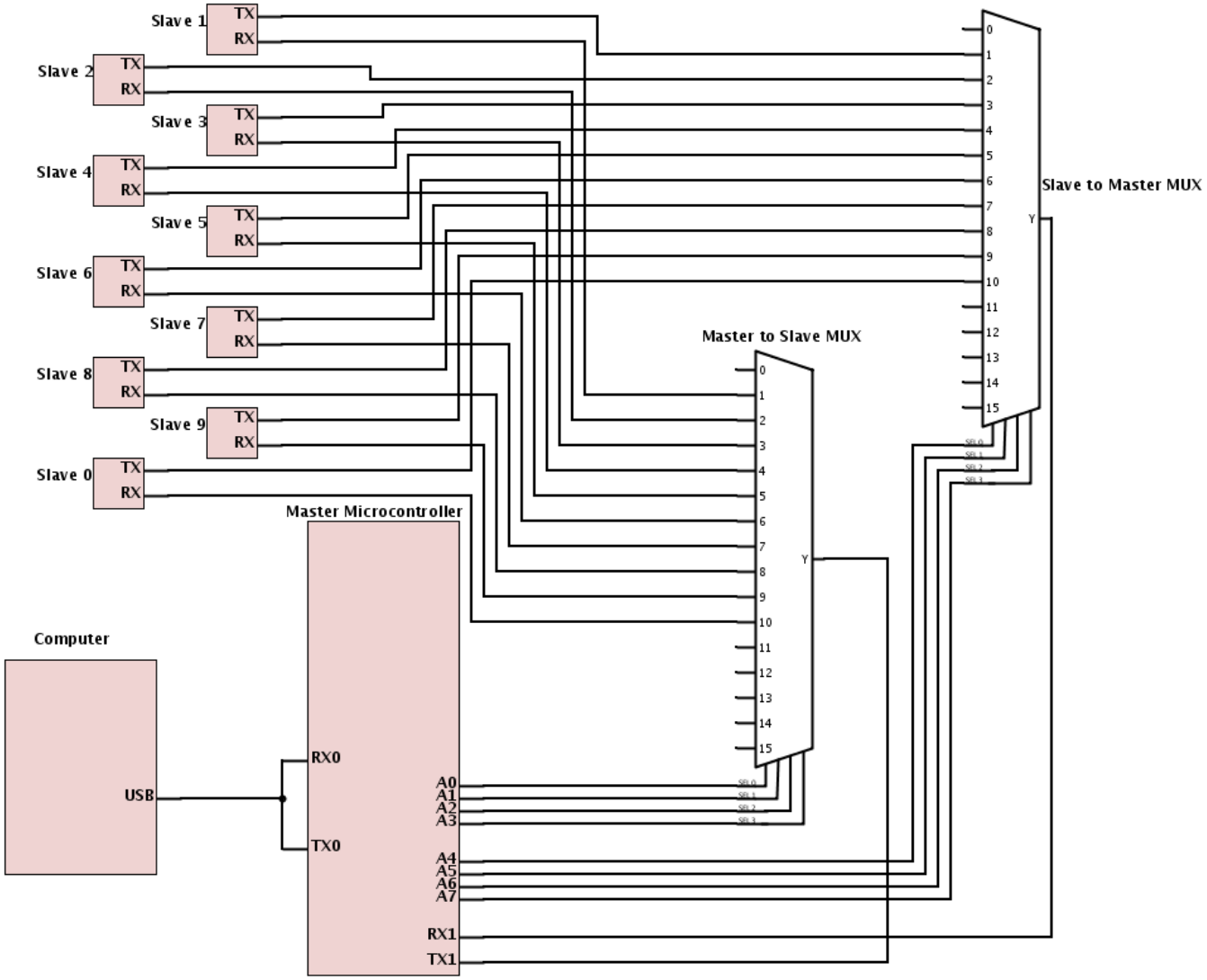
Part	Quan.	Part #
Index Distal Phalanx	1	Rapid-Prototyped
Index Intermediate Phalanx	1	Rapid-Prototyped
Index Proximal Phalanx	1	Rapid-Prototyped
Index Knuckle	1	Rapid-Prototyped
Middle Distal Phalanx	1	Rapid-Prototyped
Middle Intermediate Phalanx	1	Rapid-Prototyped
Middle Proximal Phalanx	1	Rapid-Prototyped
Middle Knuckle	1	Rapid-Prototyped
Ring Distal Phalanx	1	Rapid-Prototyped
Ring Intermediate Phalanx	1	Rapid-Prototyped
Ring Proximal Phalanx	1	Rapid-Prototyped
Ring Knuckle	1	Rapid-Prototyped
Pinky Distal Phalanx	1	Rapid-Prototyped
Pinky Intermediate Phalanx	1	Rapid-Prototyped
Pinky Proximal Phalanx	1	Rapid-Prototyped
Pinky Knuckle	1	Rapid-Prototyped
Palm Top Cover	1	Rapid-Prototyped
Palm Middle Carriage	1	Rapid-Prototyped
Palm Bottom Carriage	1	Rapid-Prototyped
Palm Bottom Cover	1	Rapid-Prototyped
Wrist Top	1	Rapid-Prototyped
Wrist Bottom	1	Rapid-Prototyped

Appendix D: Electrical Diagrams

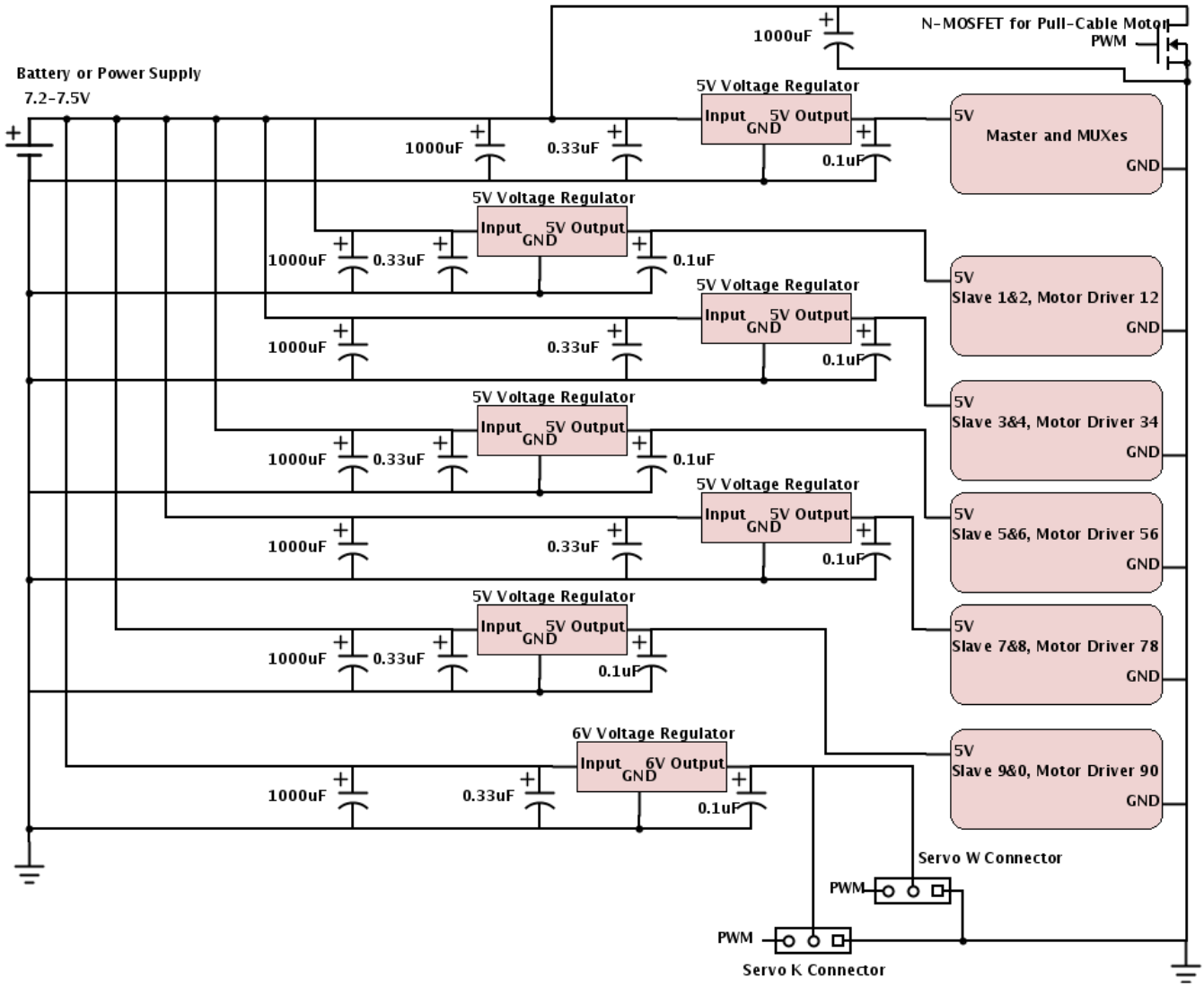
D.1 Clock Diagram



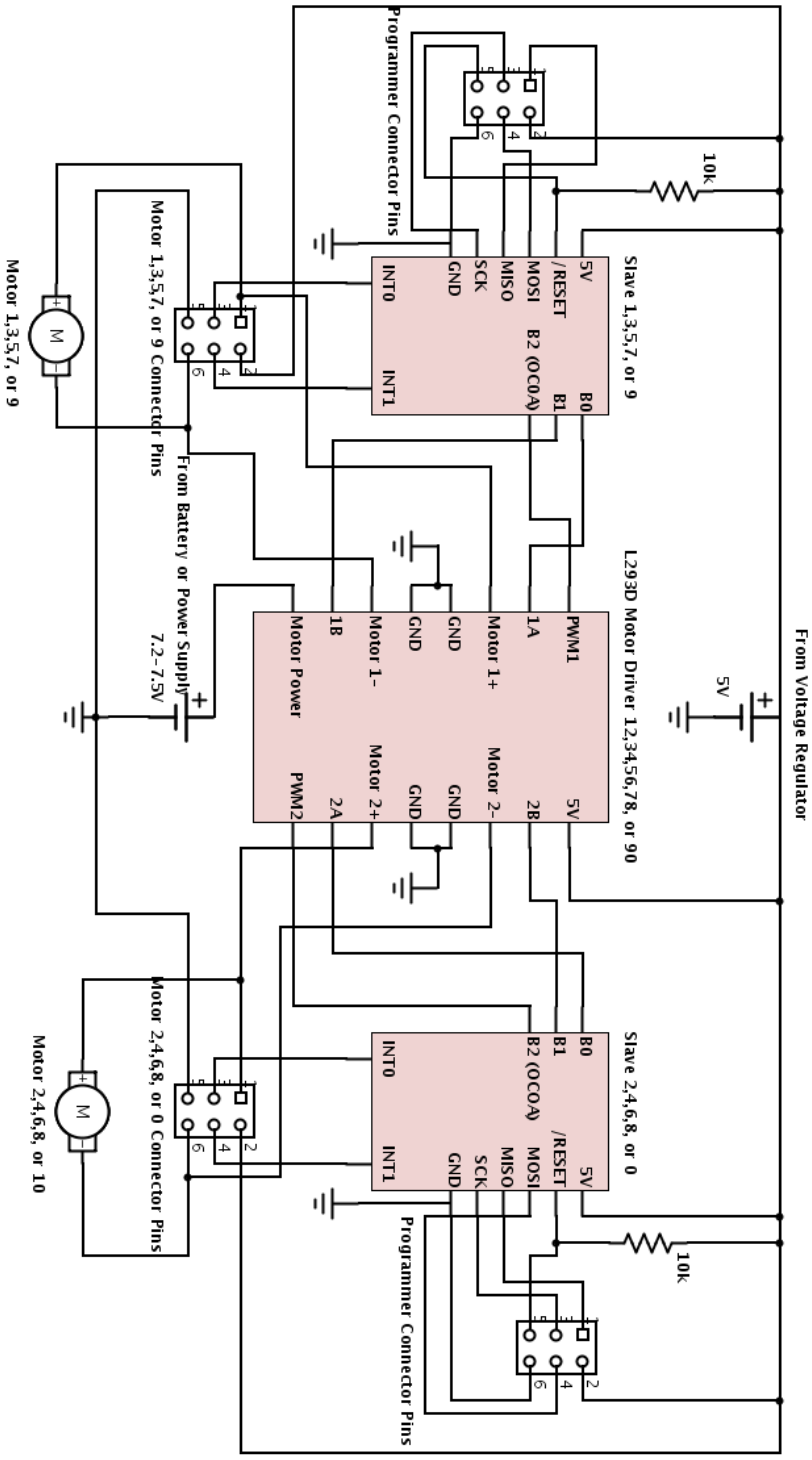
D.2 Communication Diagram



D.3 Power Diagram



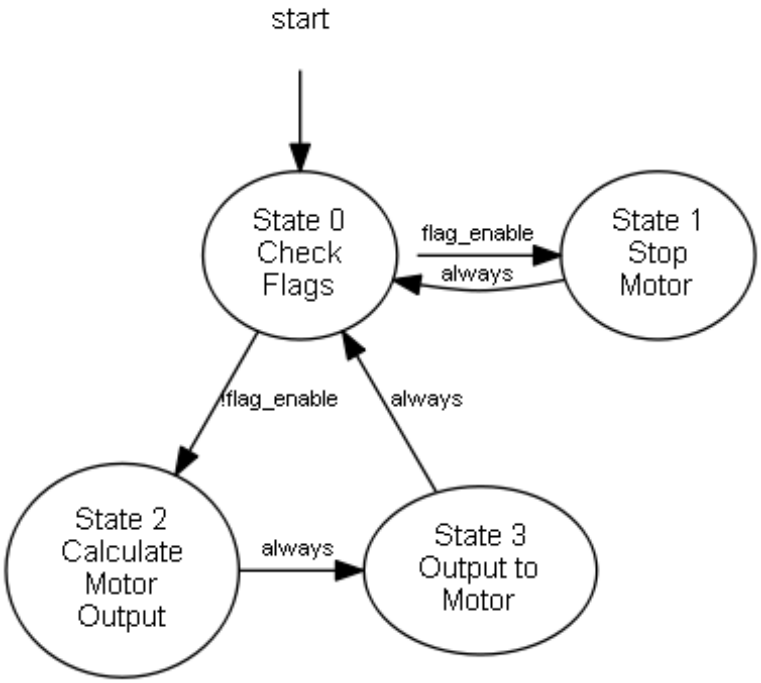
D.4 Slave Output Diagram



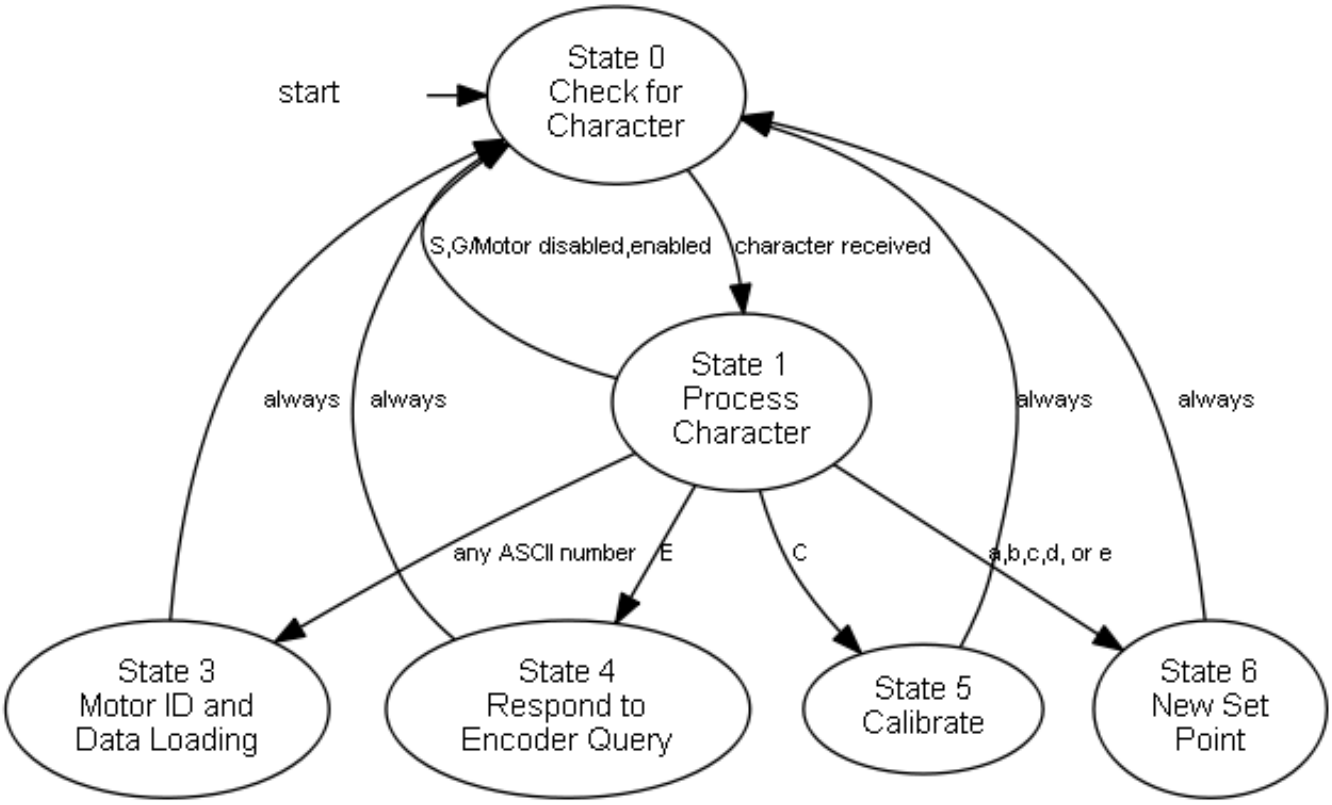
Appendix E: State-Transition Diagrams

E.1 Slave

E.1.1 Motor Task

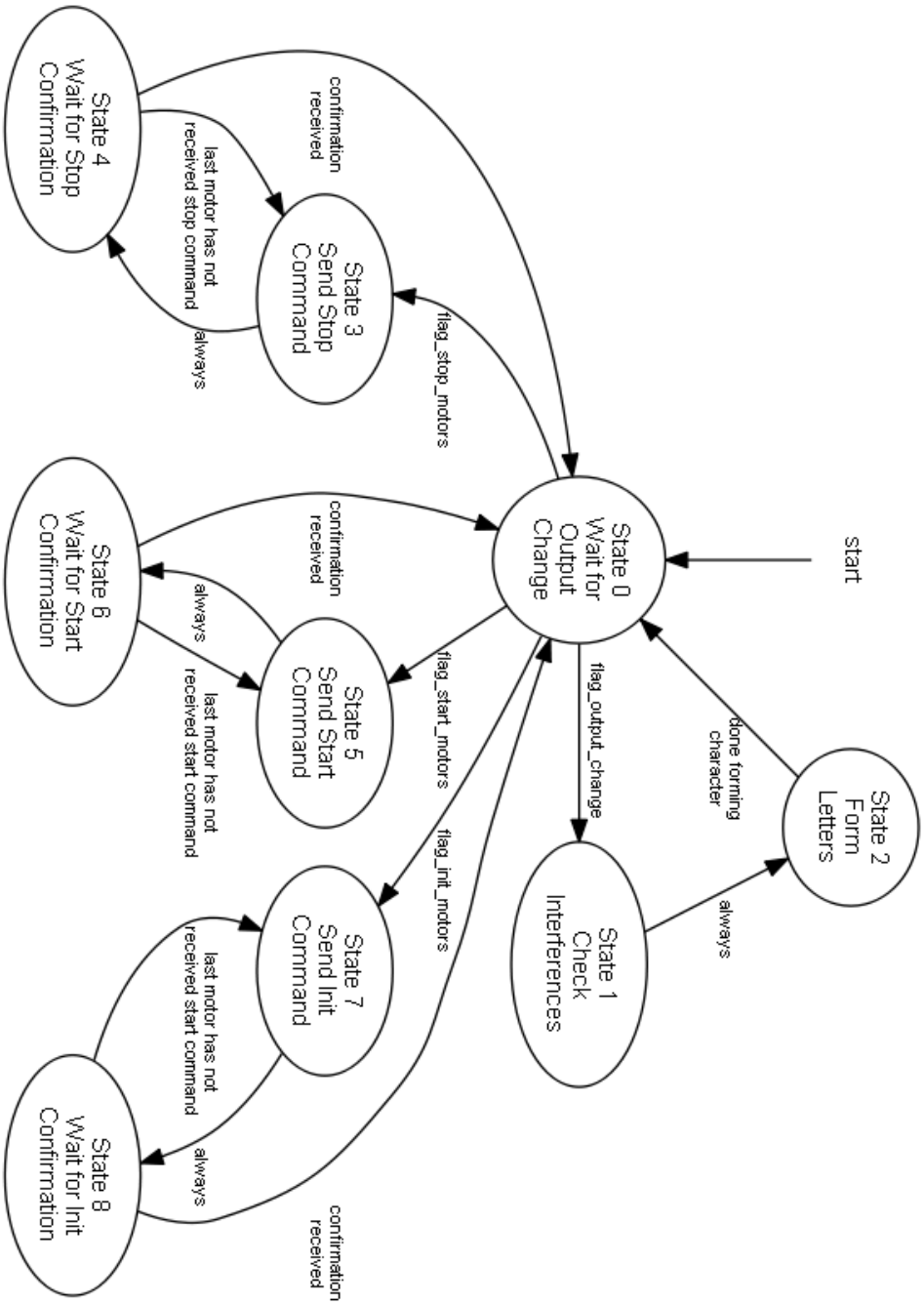


E.1.2 Data Task

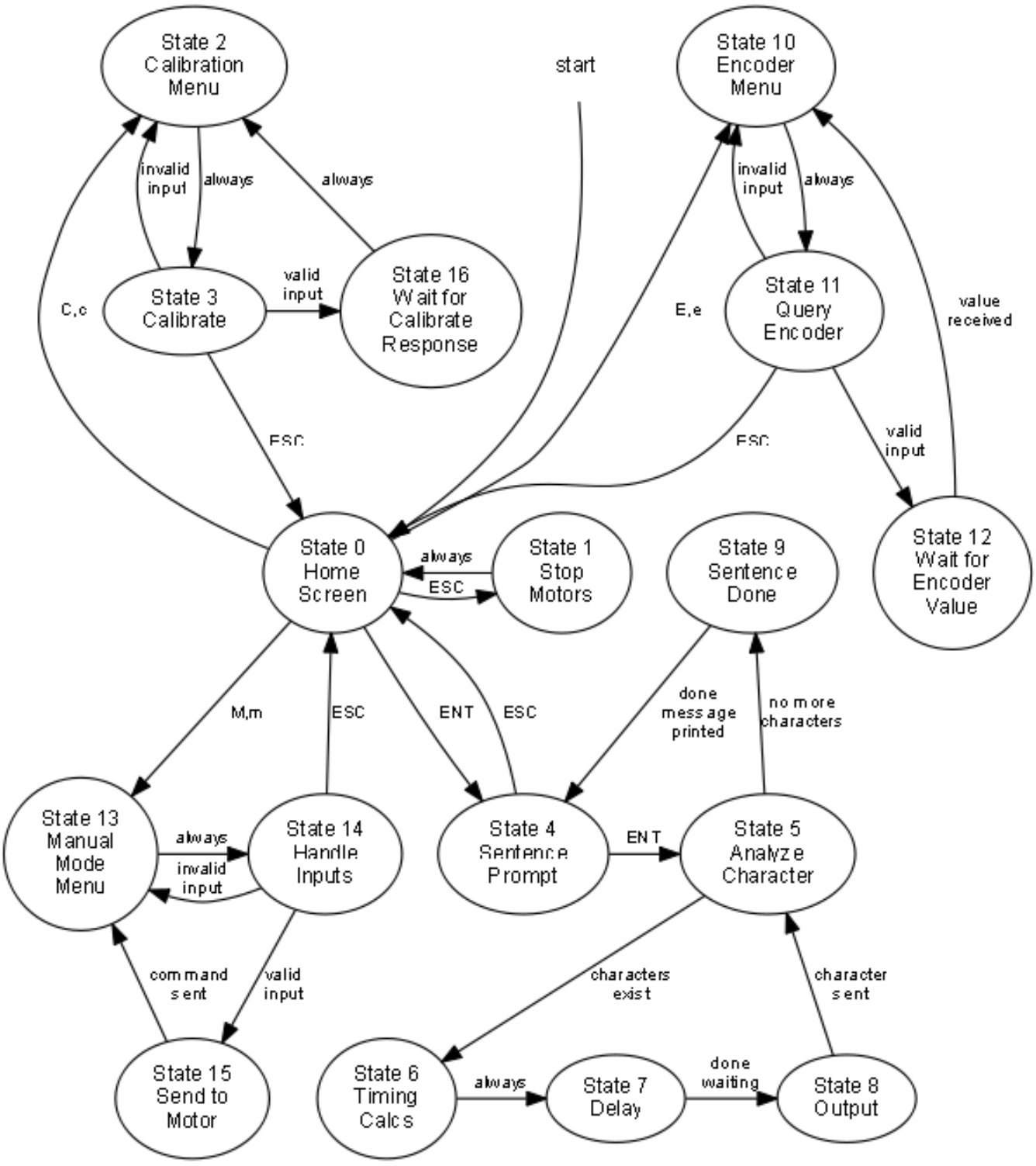


E.2 Master

E.2.1 Output Task



E.2.2 User Interface Task



Appendix F: How to Operate the Hand

Any RS232 serial port terminal program will work. This guide will use PuTTY as an example. For Windows download PuTTY from <http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html> to a folder where you can easily find it. No installation is needed. The entire program is contained in putty.exe. Linux users should install PuTTY from the software repository.

Connect the Sparkfun USB to Serial breakout board to a computer using a USB A (computer side) to Mini-B (breakout board side) cable. Make sure this is a data cable and not a charging cable. Make sure this is not a USB Micro-B cable that looks similar to a Mini-B cable. See Figure F.1 for the correct shape.

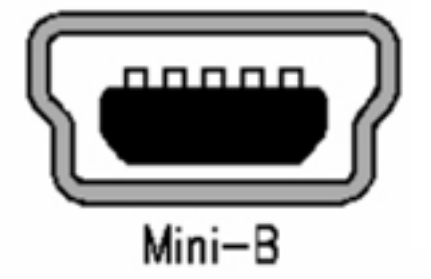


Figure F.1: Shape of the USB Mini-B plug

Verify that the board for the hand is connected to a power supply.

Determine the computer port that the breakout board connected to. This is ttyUSB0 (or ttyUSB#) on Linux-based machines or COM# on Windows machines. To locate this number in Windows, open Device Manager by searching the Start Menu (see Figure F.2).

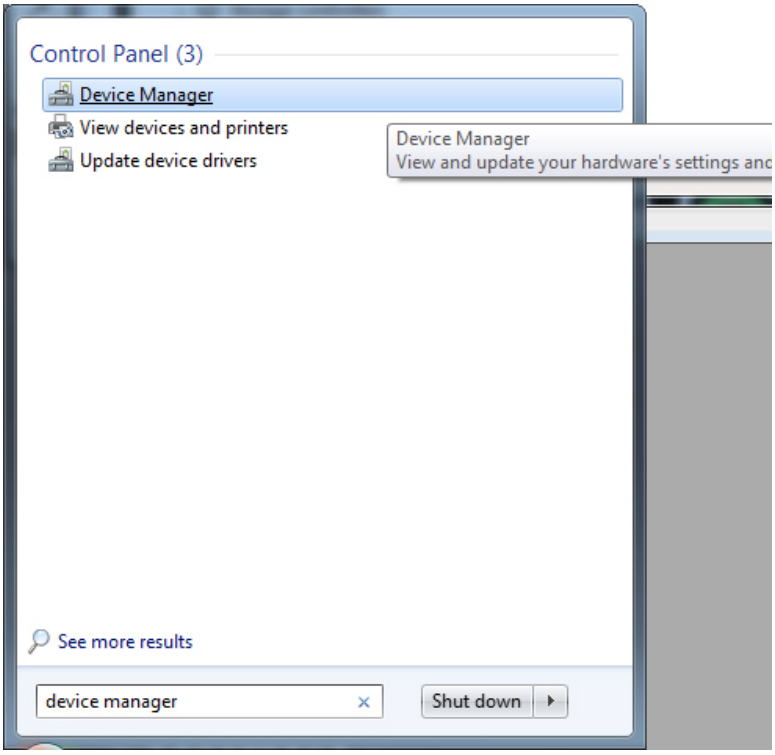


Figure F.2: Locating the Device Manager in the Start Menu

Locate "USB Serial Port" under Ports (COM & LPT). Make a note of which COM# port has been assigned to this device. In the example shown in Figure F.3, COM8 is the assigned port.

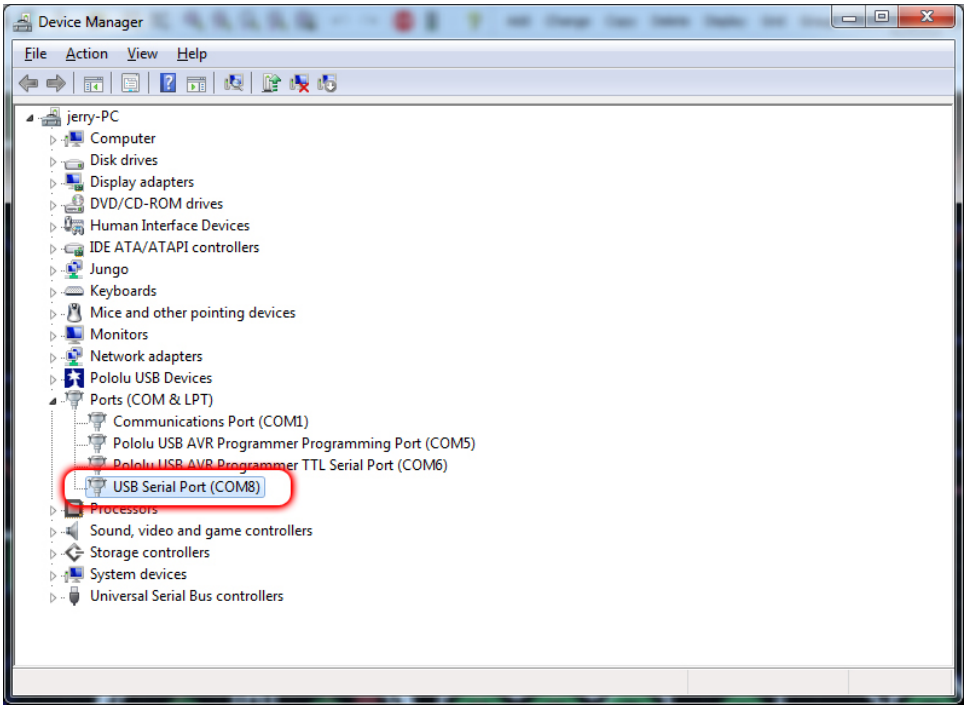


Figure F.3: Identifying the COM number using the Device Manager (COM8 in this case)

Open putty.exe. Click on Run if the security warning pops up (see Figure F.4).

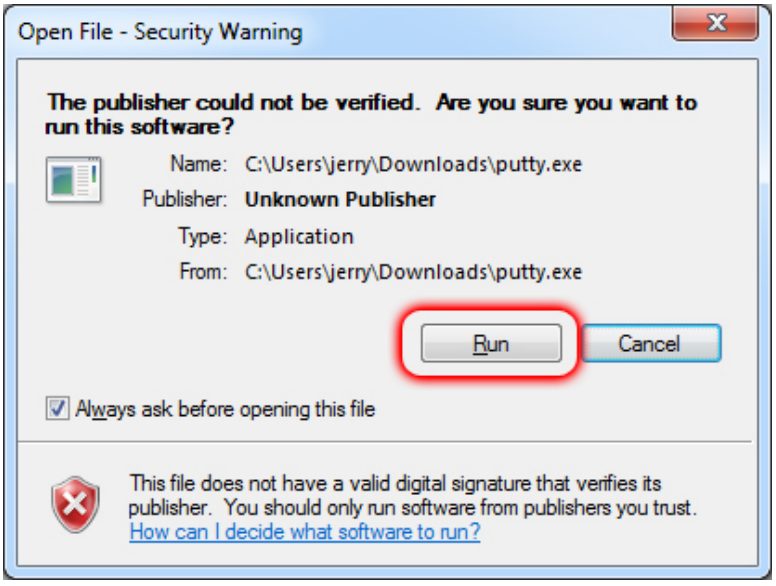


Figure F.4: PuTTY Security Warning

The main configuration menu should pop up next. Make sure you are in the Session menu (see Figure F.5).

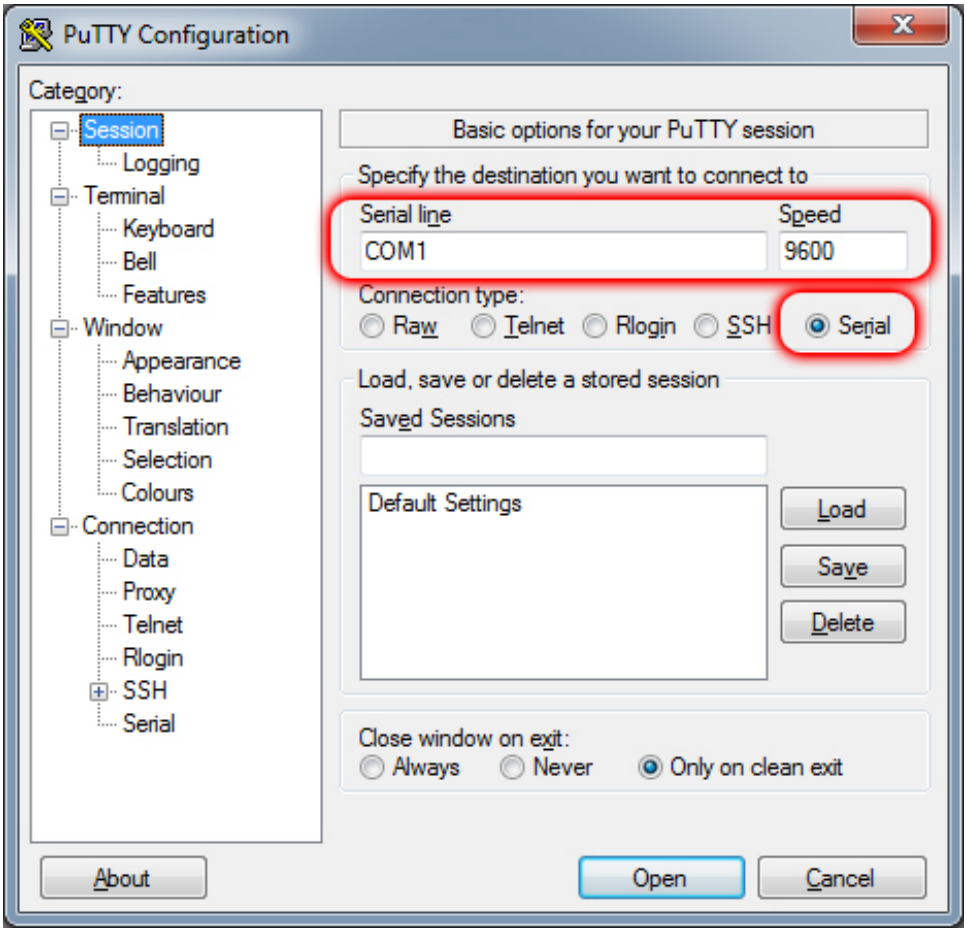


Figure F.5: PuTTY Main Session Menu

1. Change the Connection type to "Serial".

2. Change the Serial line to the COM# number identified in the Device Manager or /dev/ttyUSB0 (or similarly numbered port) on Mac/Linux.

3. Change the Speed to 9600 if not already set by default.

Switch to the Terminal menu (see Figure F.6).

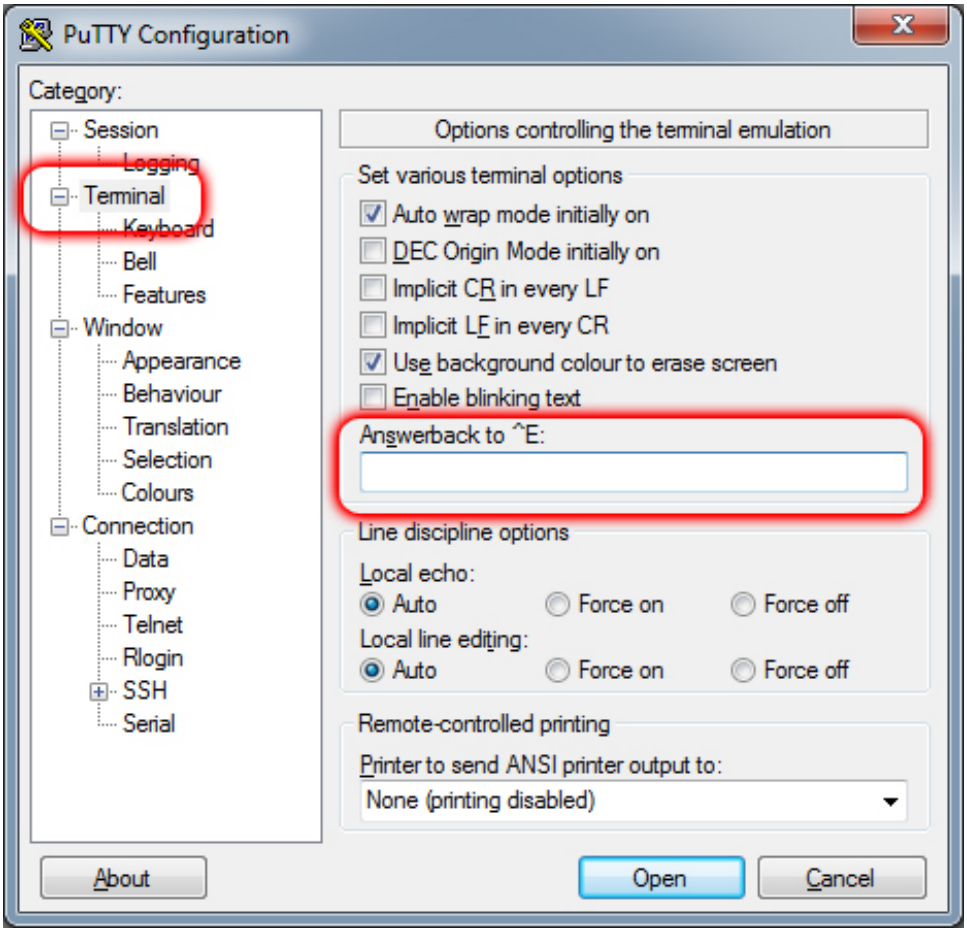


Figure F.6: PuTTY Terminal Menu

Make sure the Answerback to Ê field is blank (remove "PuTTY" if it is the default text). This prevents the terminal from printing a particular phrase to the terminal upon receiving a particular ASCII character value.

Switch to the Connection >Serial menu (see Figure F.7).

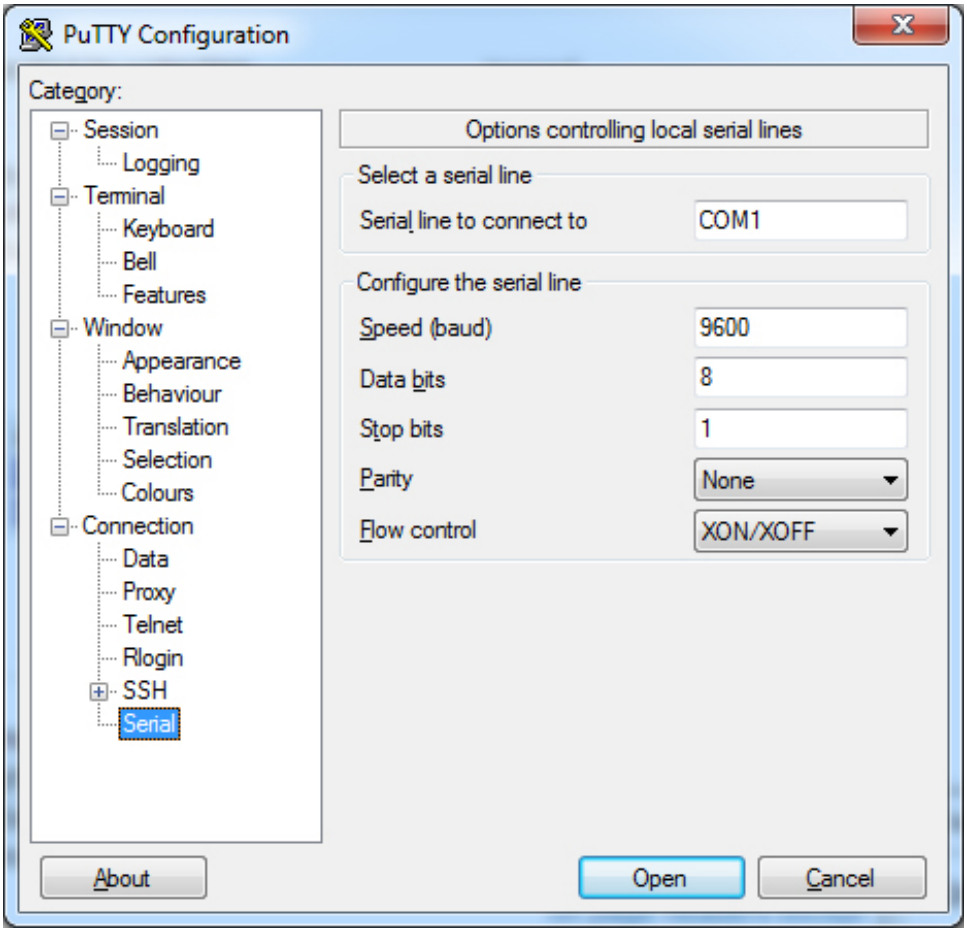


Figure F.7: PuTTY Connection >Serial Menu

1. Make sure the Serial line to connect to matches the same COM# port identified in the Session menu.

2. Change the Speed to 9600 if not already set by default.

3. Change the Data bits to 8 if not already set by default.

4. Change the Stop bits to 1 if not already set by default.

5. Change the Parity to None if not already set by default.

6. Change the Flow control to XON/XOFF if not already set by default.

Return to the Session menu and assign a name to this configuration of settings in the Saved Sessions field so they do not have to be entered over and over again. Press the Save button after assigning a name to save the settings to the list under Default Settings. Double click on the name in the list to open up a PuTTY terminal window.

Press any key not in the list of Main Menu options to reload the Main Menu (the original Menu already loaded before the terminal window did). See Figure F.8 for a screenshot of the Main Menu. Follow the different Menu options to operate the hand and navigate through the menu.

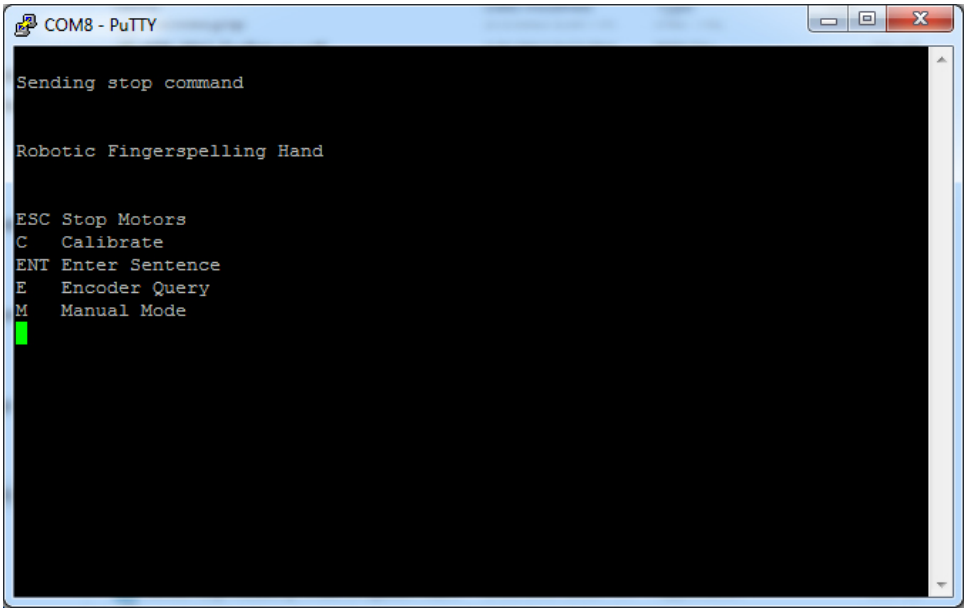


Figure F.8: Main Menu for Operating the Hand

Appendix G: Code

G.1 Slave

Code Block G.1: Main File slave.cpp

```
//=====================================================================================
/** \file controller.cc
 *  This file contains a program for an ATtiny2313 chip to control a
    single motor using proportional control.
 *  It includes interrupt service routines for monitoring the two
    quadrature encoder channels and PWM output
 *  to a motor driver chip. It also includes serial communication code to
    communicate with another
 *  microcontroller or a computer.
 *
 *  Revisions
 *  \li  04-02-2011 JV Original file for four channel quadrature decoder
 *  \li 04-05-2011 JV File changed to include complete motor control
 *  \li 04-12-2011 JV Motor control classes tested
 *
 *  License:
 *  This file released under the Lesser GNU Public License, version 2.
    This program
 *  is intended for educational use only, but it is not limited thereto.
 */
//=====================================================================================
/* Includes */

// Standard Libraries

#include <avr/io.h>      // AVR device-specific input/output definitions
#include <avr/interrupt.h> // AVR interrupt code

// Header Files

#include "motor.h"       // Motor Object
#include "serial.h"      // Serial Object
#include "angles.h"      // Angle Configuration

//=====================================================================================
/* Definitions */

#define INTERRUPT_DDR DDRD
#define INTERRUPT_PORT PORTD
#define INTERRUPT_PINR PIND
#define PIN_INT0     PIND2
```

```
#define PIN_INT1      PIND3

#define CLOCKWISE     count++
#define COUNTERCLOCKWISE count--
#define ENCODER_ERROR errors++


//============================================================================
/* Variable Definitions and Initialization */

    //uint8_t mcucsr __attribute__((section(".noinit")));

    // Serial Port
    char           character_in;     // Incoming character read by serial
        port
    char               throwaway;    // Throwaway character from when other
        chips are talked to
    unsigned char      count_input;      // Incoming count set point from
        master chip

    // Encoder Reading
    unsigned short int count = 1;        // Encoder count
    unsigned char      count_8bit = 1;   // Top 8 bits of the 10 bit
        encoder count
    unsigned char      current_reading = 0;  // Current encoder quadrature
        reading
    unsigned char      previous_reading = 0; // Last encoder quadrature
        reading
    unsigned char      errors = 0;       // Number of encoder errors

    // Configuration
    unsigned char      kp_array[10];
    unsigned char      ki_array[10];
    unsigned char      kd_array[10];
    unsigned char      set_point_angles[10];

    // Control Loop
    unsigned short int desired_count;   // Desired encoder count
    short int          control_error;    // Difference between
        encoder_count and desired_count
    unsigned char      kp;           // Proportional gain
    unsigned char      ki;           // Integral gain
    unsigned char      kd;           // Derivative gain
    long int           motor_output;     // PWM value to output to the motor
    unsigned char      set_point = 1;    // Set point (1-5) for motor
        position
    unsigned char      motor_number;     // '1'-'0' identification of which
        motor number

    // State Transition Logic
```

```
    unsigned char     state_motor = 0;  // Next state to jump into in
        motor task
    unsigned char     state_data = 0; // Next state to jump into for data
        task

    // Flags
    bool              flag_enable = false;  // Motor output enable
    bool              flag_calibrate = false; // Encoder calibration flag

    // Miscellaneous
    unsigned long     i = 0;         // Dummy counter
    unsigned char     i_angle = 0;   // Angle count

    // Objects
    motor mtr;
    serial sport;



//=============================================================================================
/* State-Transition Logic Tasks */

// Motor Task

    unsigned char motor_task(unsigned char state_motor, motor* the_motor)
    {
        mtr = *the_motor;

        switch(state_motor)
        {
            case(0):      // Check Flags
                if (!(flag_enable)) // If motor stop command issued
                {
                    state_motor = 1;  // go to state 1
                    break;
                }
                if (flag_enable)  // If motor go command issued (or no
                    overriding commands issued)
                {
                    state_motor = 2;  // go to state 2
                    break;
                }
                break;
            case(1):      // Stop Motor
                mtr.stop();    // Activate brake
                state_motor = 0;  // go to state 0
                break;
            case(2):      // Calculate Motor Output

                // Calculate control loop error
```

```
            control_error = count - desired_count;

            // Calculate value and trim to 0-255 range
            if (control_error > 1)
                motor_output = (long) (kp * control_error * 255) / 768;
            else if (control_error < -1)
                motor_output = (long) (0xFF * control_error * 255) / 768;
            else
                motor_output = 0;
            if (motor_output > 255)
                motor_output = 255;

            state_motor = 3;  // go to state 3
            break;
        case(3):      // Output to Motor
            // Set direction
            if (control_error > 0)
            {
                mtr.d1();
                mtr.output( (unsigned char) motor_output);
            }
            else if (control_error < 0)
            {
                mtr.d0();
                mtr.output( (unsigned char) motor_output);
            }
            else
                mtr.stop();

            state_motor = 0;  // Always return to state 1
            break;
        default:
            state_motor = 0;
            break;
    }
    return(state_motor);
}

// Data Task

    unsigned char data_task(unsigned char state_data, serial* serial_port,
        motor* the_motor)
    {
        sport = *serial_port;
        mtr = *the_motor;

        switch(state_data)
        {
            case(0):      // Check for Character
                if(sport.check_for_char())
                {
```

```
            state_data = 1; // If character received go to state 1
        }
        else
        {
            state_data = 0; // If not remain in state 0
        }
        break;
    case(1):      // Process Character

        // Interpret character
        switch(character_in)
        {
            // a,b,c,d,e define set points
            case('a'):
            case('b'):
            case('c'):
            case('d'):
            case('e'):
                switch(character_in)
                {
                    case('a'):
                        set_point = 1;
                        break;
                    case('b'):
                        set_point = 2;
                        break;
                    case('c'):
                        set_point = 3;
                        break;
                    case('d'):
                        set_point = 4;
                        break;
                    case('e'):
                        set_point = 5;
                        break;
                    default:
                        break;
                }
                //sport.send('A');    // Confirm command reception
                state_data = 6;
                break;
            // S,G disable and enable the motor
            case('S'): // Stop Motor
                flag_enable = false;  // Disable motor
                state_data = 0;   // Go to state 1
                sport.send('s');      // Confirm command reception
                break;
            case('G'): // Go (enable motor)
                flag_enable = true;   // Enable motor
                state_data = 0;       // Go to state 1
                sport.send('g');      // Confirm command reception
```

```
            break;
        // C clears the encoder count to calibrate the motor
            position
        case('C'): // Calibrate
            flag_calibrate = !flag_calibrate; // Toggle
                calibration flag
            state_data = 5;                 // Go to state 5
            sport.send('c');                // Confirm command
                reception
            break;
        case('1'): // Identify motor
        case('2'):
        case('3'):
        case('4'):
        case('5'):
        case('6'):
        case('7'):
        case('8'):
        case('9'):
            motor_number = character_in - 0x30;
            state_data = 3;
            break;
        case('0'):
            motor_number = 10;
            state_data = 3;
            break;
        case('E'): // Encoder Query
            state_data = 4;
            break;
        default:
            state_data = 0; // Return to state 1 if character is
                unclear
            break;
    }
    break;
case(3):      // Motor Identification and data loading
    // Load angle data
    for (i_angle = 0; i < 5; i++)
    {
        set_point_angles[i_angle] =
            angles[i_angle][motor_number-1];
    }

    // Load gain data
    kp = kp_array[motor_number-1];

    // Send confirmation back to master
    sport.send('!');

    state_data = 0;
    break;
```

```
            case(4):      // Respond to Encoder Query
                count_8bit = (unsigned char) (count << 2);
                sport.send(count_8bit);
                state_data = 0;
                break;
            case(5):      // Calibrate
                if (!flag_calibrate)  // If the calibration flag has been
                    turned off
                {
                    count = 1; // Clear count
                }
                state_data = 0; // Always return to state 1
                break;
            case(6):      // New set point
                desired_count = set_point_angles[set_point-1];
                state_data = 0;
                break;
            default:
                state_data = 0;
                break;
        }
        return(state_data);
    }

//=============================================================================================
/* Main Function */

// Initialize variables

int main(void)
{


// Setup

    // Create objects
    motor mtr; // Create motor
    serial sport;  // Create serial port


    // Setup Encoder Data Directions
    INTERRUPT_DDR &= ~(1 << PIN_INT0); // Input
    INTERRUPT_DDR &= ~(1 << PIN_INT1); // Input

    // Enable interrupts on INT0, INT1, and PCINT2

        // Enable interrupt on both rising and falling edges for both pins
        MCUCR |= (1 << ISC10); // ISC11 = 0, ISC10 = 1
        MCUCR |= (1 << ISC00); // ISC01 = 0, ISC00 = 1

        // Enable interrupts on INT0, INT1, and PCIE
```

```
        GIMSK |= (1 << INT0);
        GIMSK |= (1 << INT1);
        GIMSK |= (1 << PCIE);

        // Enable interrupts on PCINT2
        PCMSK |= (1 << PCINT2); // Write 1 to PCINT2 bit of PCMSK register

    // Turn on interrupts
    sei();

    // Motor Data
/*  for (i = 1; i<10; i++)
    {
        kp_array[i] = 4;
    }
*/
    bool data_sent = false;

// Loop

    while(true) // loop forever between these two tasks
    {
        if (!data_sent)
        {
            sport.send('A');
            data_sent = true;
        }

        state_motor = motor_task(state_motor, &mtr);
        state_data = data_task(state_data, &sport, &mtr);
    }
    return(0);
}

//=============================================================================================
/* Interrupt Service Routines */


// Interrupt for Encoder Channel A
ISR(INT0_vect)
{

    // Store last reading to old variable
    previous_reading = current_reading;

    // Take in new reading
    current_reading = ((PIND & 0b00000100) >> 1) | ((PIND & 0b00001000) >>
        3); // (A << 1) | B

    // Evaluate Reading
    switch(current_reading)
```

```
{
    case(0):              // Current value == 00
        switch(previous_reading)
        {
            case(1):      // 01 to 00
                CLOCKWISE ;
                break;
            case(2):      // 10 to 00
                COUNTERCLOCKWISE ;
                break;
            default:      // 11 to 00
                ENCODER_ERROR ;
                break;
        }
        break;
    case(1):              // Current value == 01
        switch(previous_reading)
        {
            case(3):      // 11 to 01
                CLOCKWISE ;
                break;
            case(0):      // 00 to 01
                COUNTERCLOCKWISE ;
                break;
            default:      // 10 to 01
                ENCODER_ERROR ;
                break;
        }
        break;
    case(2):              // Current value == 10
        switch(previous_reading)
        {
            case(0):      // 00 to 10
                CLOCKWISE ;
                break;
            case(3):      // 11 to 10
                COUNTERCLOCKWISE ;
                break;
            default:      // 01 to 10
                ENCODER_ERROR ;
                break;
        }
        break;
    case(3):              // Current value == 11
        switch(previous_reading)
        {
            case(2):      // 10 to 11
                CLOCKWISE ;
                break;
            case(1):      // 01 to 11
                COUNTERCLOCKWISE ;
```

```
                break;
            default:      // 00 to 11
                ENCODER_ERROR ;
                break;
        }
        break;
    default:
        break;
  }
}

// Interrupt for Encoder Channel B
ISR(INT1_vect, ISR_ALIASOF(INT0_vect)); // Duplicate code from encoder
    channel A
```

Code Block G.2: Serial Header serial.h

```
//=============================================================================
/** \file motor.h
 *  This file contains a header for an ATtiny2313 motor driver. The driver
     will output a PWM signal to a
 *  L293D motor driver chip along with two directional signals.
 *
 *  Revised:
 *    \li 04-09-2011 JV Original file.
 *
 *  License:
 *  This file released under the Lesser GNU Public License, version 2.
     This program
 *  is intended for educational use only, but it is not limited thereto.
 */
//=============================================================================
/// This define prevents this .h file from being included more than once
    in a .cc file
#ifndef _SERIAL_H_
#define _SERIAL_H_

//=============================================================================
/* Definitions */

#define CPU_FREQ_Hz 20000000
#define BAUD_RATE 9600
#define BAUD_DIV  (((CPU_FREQ_Hz) / (16UL * (BAUD_RATE))))

//=============================================================================

//-----------------------------------------------------------------------------
/** This class sets up a serial class for the ATtiny 2313
 */

class serial
{
    protected:
        /// This is a pointer to the data register used by the UART
        volatile unsigned char* p_UDR;

        /// This is a pointer to the status register used by the UART
        volatile unsigned char* p_USR;

        /// This is a pointer to the control register used by the UART
        volatile unsigned char* p_UCR;

        /// This bitmask identifies the bit for data register empty, UDRE
        unsigned char mask_UDRE;

        /// This bitmask identifies the bit for receive complete, RXC
        unsigned char mask_RXC;
```

```
        /// This bitmask identifies the bit for transmission complete, TXC
        unsigned char mask_TXC;
    public:
        /// The constructor sets up the port with the given baud rate and
            port number.
        serial (void);

        /// This method sends a byte out
        void send(unsigned char);

        /// This method checks if the serial port is ready to transmit data.
        bool ready_to_send (void);

        /// This method returns true if the port is currently sending a
            character out.
        bool is_sending (void);

        /// This method returns true if a character has been read by the
            serial port.
        bool check_for_char (void);

        /// This method returns the latest character received by the serial
            port.
        char getchar (void);
};

//=============================================================================

#endif
```

Code Block G.3: Serial Class serial.cpp

```
//=============================================================================
/** \file serial.cc
 *  This file contains a program for an ATtiny2313 serial driver.
 *
 *  Revised:
 *    \li 04-03-2006 JRR For updated version of compiler
 *    \li 06-10-2006 JRR Ported from C++ to C for use with some C-only
    projects; also
 *        serial_avr.h now has defines for compatibility among lots of AVR
    variants
 *    \li 08-11-2006 JRR Some bug fixes
 *    \li 03-02-2007 JRR Ported back to C++. I've had it with the
    limitations of C.
 *    \li 04-16-2007 JO Added write (unsigned long)
 *    \li 07-19-2007 JRR Changed some character return values to bool,
    added m324p
 *    \li 01-12-2008 JRR Added code for the ATmega128 using USART number 1
    only
 *    \li 02-14-2008 JRR Split between base_text_serial and rs232 files
 *    \li 05-31-2008 JRR Changed baud calculations to use CPU_FREQ_MHz from
    Makefile
 *    \li 06-01-2008 JRR Added getch_tout() because it's needed by 9Xstream
    modems
 *    \li 07-05-2008 JRR Changed from 1 to 2 stop bits to placate finicky
    receivers
 *    \li 12-22-2008 JRR Split off stuff in base232.h for efficiency
 *    \li 01-30-2009 JRR Added class with port setup in constructor
 *    \li 04-09-2009 JRR Changed to a simpler baud rate calculation formula
 *    \li 04-08-2011 JV New file based on Dr. Ridgely's base232.h file
 *
 *  License:
 *  This file released under the Lesser GNU Public License, version 2.
    This program
 *  is intended for educational use only, but it is not limited thereto.
 */
//=============================================================================

#include <avr/io.h>
#include "serial.h"

/** This constructor sets up a USART serial port for the ATtiny2313.
 */

serial::serial (void)
{
    p_UDR = &UDR;     // Input/Output data register
    p_USR = &UCSRA; // Control and Status Register A
    p_UCR = &UCSRB; // Control and Status Register B

    // Setup USART Control and Status Register B (UCSRB)
```

```
    UCSRB = (1 << RXEN) | (1 << TXEN);  // Enable RX and TX

    // Setup USART Control and Status Register C (UCSRC)
    UCSRC = (1 << UCSZ1) | (1 << UCSZ0); // Set UCSZ bits to 8 bit
        character size

    // Calculate baud rate divisor
    UBRRH = (unsigned char)(BAUD_DIV >> 8);
    UBRRL = (unsigned char) BAUD_DIV;

    // Setup USART Control and Status Register A (UCSRA)
    UCSRA |= U2X; // Double Speed

    // Create Masks
    mask_UDRE = (1 << UDRE);
    mask_RXC = (1 << RXC);
    mask_TXC = (1 << TXC);
}

/** This method will send data out the serial port.
 *  @param data_out The byte to send out.
 */
void serial::send (unsigned char data_out)
{
    *p_UDR = data_out;
}

/** This method checks if the serial port transmitter is ready to send
    data. It
 *  tests whether transmitter buffer is empty.
 *  @return True if the serial port is ready to send, and false if not
 */
bool serial::ready_to_send (void)
{
    if (*p_USR & mask_UDRE)
        return (true);

    return (false);
}

/** This method checks if the serial port is currently sending a
    character. Even if the
 *  buffer is ready to accept a new character, the port might still be
     busy sending the
 *  last one; it would be a bad idea to put the processor to sleep before
     the character
 *  has been sent.
 *  @return True if the port is currently sending a character, false if
     it's idle
 */
bool serial::is_sending (void)
```

```
{
    if (*p_USR & mask_TXC)
        return (false);
    else
        return (true);
}

/** This method checks if the serial port has received a character and
    stored it in the buffer.
 *  @return True if the port has a character in the buffer, false if it
     does not
 */
bool serial::check_for_char (void)
{
    if (*p_USR & mask_RXC)
        return (true);
    else
        return (false);
}

/** This method gets one character from the serial port, if one is there.
    If not, it
 *  waits until there is a character available. This can sometimes take a
     long time
 *  (even forever), so use this function carefully. One should almost
     always use
 *  check_for_char() to ensure that there's data available first.
 *  @return The character which was found in the serial port receive buffer
 */
char serial::getchar (void)
{
    // Wait until there's something in the receiver buffer
    while ((*p_USR & mask_RXC) == 0);

    // Return the character retreived from the buffer
    return (*p_UDR);
}
```

Code Block G.4: Motor Header motor.h

```
//=============================================================================
/** \file motor.h
 *  This file contains a header for an ATtiny2313 motor driver that
    outputs to a L293D motor driver chip. The
 *  ATtiny2313 will output a PWM signal and two directional signals to
    determine the H-bridge inputs.
 *
 *  Revised:
 *    \li 04-09-2011 JV Original file
 *
 *  License:
 *  This file released under the Lesser GNU Public License, version 2.
    This program
 *  is intended for educational use only, but it is not limited thereto.
 */
//=============================================================================
/// This define prevents this .h file from being included more than once
   in a .cc file
#ifndef _MOTOR_H_
#define _MOTOR_H_

//=============================================================================
/* Definitions */

#define MOTOR_PORT PORTB     ///< Name of port connected to motor output
   pins
#define MOTOR_DDR DDRB          ///< Data direction register for motor
   output pins
#define PIN_PWM PINB2    ///< PWM output pin
#define PIN_INA PINB1    ///< Direction input A
#define PIN_INB PINB0    ///< Direction input B

//=============================================================================
/* Class Definition */

//-----------------------------------------------------------------------------
/** This class sets up a motor class for the ATtiny 2313
 */

class motor
{
   public:
      /// The constructor sets up the port with the given baud rate and
         port number.
      motor (void);

      /// This method performs an emergency stop
      void stop (void);

      /// This method sets the motor to spin in direction 0
```

```
        void d0 (void);

        /// This method sets the motor to spin in direction 1
        void d1 (void);

        /// This method outputs a PWM value at a given direction
        void output (unsigned char);
};

//=============================================================================

#endif
```

Code Block G.5: Motor Class motor.cpp

```
//=============================================================================
/** \file motor.h
 *  This file contains program for an ATtiny2313 motor driver that outputs
    to a L293D motor driver chip. The
 *  ATtiny2313 will output a PWM signal and two directional signals to
    determine the H-bridge inputs.
 *
 *  Revised:
 *    \li 04-09-2011 JV Original file
 *
 *  License:
 *  This file released under the Lesser GNU Public License, version 2.
    This program
 *  is intended for educational use only, but it is not limited thereto.
 */
//=============================================================================

#include <avr/io.h>
#include "motor.h"

//-----------------------------------------------------------------------------
/** This constructor sets up the ATtiny2313 for motor driving.
 */

motor::motor (void)
{
    // Set up registers for Output Compare on OC0A

        // Set up Timer/Counter Control Register A
        TCCR0A = (1 << COM0A1); // Clear OC0A on Compare Match, set OC0A at
            timer max
        TCCR0A |= (1 << WGM01); // Fast PWM Mode. Count from 0 to 255
        TCCR0A |= (1 << WGM00);

        // Set up Timer/Counter Control Register B
        TCCR0B = (1 << CS02); // Divide clock by prescaler 256

        // Clear Output Compare Register
        OCR0A = 0; // Clear motor output pwm register

    // Set up pins for output

        // Set data directions
        MOTOR_DDR |= (1 << PIN_PWM); // Set all three pins to outputs
        MOTOR_DDR |= (1 << PIN_INA);
        MOTOR_DDR |= (1 << PIN_INB);

        // Set both directional pins off
        MOTOR_PORT &= ~(1 << PIN_INA);
        MOTOR_PORT &= ~(1 << PIN_INB);
```

```
}

void motor::stop (void)
{
    // Both pins high
    MOTOR_PORT |= (1 << PIN_INA);
    MOTOR_PORT |= (1 << PIN_INB);
}

void motor::d0 (void)
{
    // A high; B low
    MOTOR_PORT |= (1 << PIN_INA);
    MOTOR_PORT &= ~(1 << PIN_INB);
}

void motor::d1 (void)
{
    // A low; B high
    MOTOR_PORT &= ~(1 << PIN_INA);
    MOTOR_PORT |= (1 << PIN_INB);
}

void motor::output (unsigned char duty_cycle)
{
    // Set duty cycle
    OCR0A = duty_cycle;
}
```

Code Block G.6: Motor Preset Data angles.h

```
#ifndef _ANGLES_H_
#define _ANGLES_H_

    unsigned char     angles[5][10] = { {1,1,1,1,1,1,1,1,1,1},
                                        {65,65,65,65,65,65,65,65,65,65},
                                        {96,96,96,96,96,96,96,96,96,96},
                                        {128,128,128,128,128,128,128,128,128,128},
                                        {192,192,192,192,192,192,192,192,192,192}
                                            };

#endif
```

G.2 Master

Code Block G.7: Main File master.cpp

```
//============================================================================
/** \file mototest.cpp
 *   This file contains a program to test the ME 405 board's motor driver.
    It makes a
 *   motor move at speeds determined by the value read from a
    potentiometer connected
 *   to the AVR's analog inputs.
 *
 *  Revisions
 *   \li 01-05-2008 JRR Original file
 *   \li 02-03-2008 JRR Various cleanup, tested on new ME 405 boards
 *   \li 01-15-2008 JRR Changed to new file/directory layout with ./lib
    and *.cpp
 *   \li 01-19-2011 JRR Updated calls to newer version of stl_task
    constructor
 *
 *  License:
 *   This file released under the Lesser GNU Public License, version 2.
    This program
 *   is intended for educational use only, but it is not limited thereto.
 */
//============================================================================

                                    // System headers included with < >
#include <stdlib.h>                 // Standard C library
#include <avr/io.h>                 // Input-output ports, special
    registers
#include <avr/interrupt.h>          // Interrupt handling functions

                                    // User written headers included
                                       with " "
#include "lib/queue.h"              // Queue class used for character
    buffer
#include "servo.h"                  // Servo class
#include "slave_picker.h"           // The class that sets the
    multiplexer pins
#include "character.h"
#include "character_database.h"     // The class that stores all
    character info
#include "lib/rs232int.h"           // Serial port header
#include "lib/global_debug.h"       // Header for serial debugging port
//#include "motor.h"                   // Class containing all motors
#include "lib/stl_timer.h"          // Microsecond-resolution timer
#include "lib/stl_task.h"           // Base class for all task classes
#include "task_output.h"            // The task that outputs all
    commands to the motor controllers
```

```
#include "task_user.h"                  // The task that listens to the
    user


//-------------------------------------------------------------------------------
/** The main function is the "entry point" of every C program, the one
    which runs first
 *  (after standard setup code has finished). For mechatronics programs,
     main() runs an
 *  infinite loop and never exits.
 */

int main ()
{
    volatile unsigned int dummy = 0;    // Delay loop kind of counter

    // Create objects

            // Create a serial port object. Messages will be printed to
                this port, which
            // should be hooked up to a dumb terminal program like minicom
                on a PC
            rs232 sport_slave (9600, 1);
            rs232 sport_comp (9600, 0);
            set_glob_debug_port (&sport_comp);

            // Create a slave picker
            slave_picker the_slave_picker(&sport_comp);

            // Create a character database
            character_database char_dbase;

            // Servos and motor databases
            servo servo_top(1);
            servo servo_bottom(2);
            //motor the_motors
                (&sport_slave,&the_slave_picker,&servo_top,&servo_bottom,&sport_comp);

            // Create a microsecond-resolution timer
            task_timer the_timer;

    // Create and set up tasks

            // Create a time stamp which holds the interval between runs of
                the motor task
            // The time stamp is initialized with a number of seconds, then
                microseconds
            time_stamp interval_time (0, 10000);

            // Set the interval to 20ms
            interval_time.set_time (0, 10000);
```

```
            task_output output_task (the_timer, interval_time, &sport_comp,
                &sport_slave, &the_slave_picker,&servo_top,&servo_bottom);

            // Set the interval a bit slower for the user interface task
            interval_time.set_time (0, 25000);

            // Create a task to read commands from the keyboard
            task_user user_task (the_timer, interval_time, &sport_comp,
                &sport_slave, &the_slave_picker, &output_task);

            // Turn on interrupt processing so the timer can work
            sei ();

    // Run the main scheduling loop, in which the tasks are continuously
        scheduled.
    // This program currently uses very simple "round robin" scheduling in
        which the
    // tasks are simply called in order. More sophisticated scheduling
        strategies
    // will be used in other more sophisticated programs
    while (true)
    {
        output_task.schedule ();
        user_task.schedule ();
    }

    return (0);
}
```

Code Block G.8: Output Task Header task_output.h

```
//*****************************************************************************
/** \file task_output.h
 *   This file contains a task class for running a user interface for the
    motor
 *   controller demonstration. It has a single-state task which just reads
    the
 *   serial port to see if the user typed anything, and acts if s/he has
    done so.
 *
 *  Revisions:
 *    \li 02-06-2008 JRR Original file
 *    \li 05-15-2008 JRR Modified to work with two motor drivers rather
    than one
 *    \li 03-08-2009 JRR Added code to test A/D converter
 *    \li 01-19-2011 JRR Updated calls to newer version of stl_task
    constructor
 *
 *  License:
 *  This file released under the Lesser GNU Public License, version 2.
    This program
 *  is intended for educational use only, but it is not limited thereto.
 */
//*****************************************************************************

//#include "motor.h"
#include "servo.h"

#ifndef _TASK_OUTPUT_H_
#define _TASK_OUTPUT_H_

#define MAX_SENTENCE_SIZE 255

#define KEY_ESCAPE       0x1B
#define KEY_ENTER        0x0D
#define KEY_BACKSPACE    0x08

//-----------------------------------------------------------------------------
/** This class contains a task which moves a motorized lever back and
   forth.
 *  WARNING: This task uses an older version of parent class stl_task, and
    its
 *           constructor parameters are out of date. Use it as an example,
    but
 *           do not attempt to just copy the parameters.
 */

class task_output : public stl_task
{
   protected:
```

```
    base_text_serial*  p_serial_comp;       ///< Pointer to serial
       device for computer
    base_text_serial*  p_serial_slave;      ///< Pointer to serial
       device for slave
    slave_picker*      p_slave_chooser;     ///< Pointer to slave
       picker for mux pins
    //motor*              p_motors;         ///< Pointer to all the
       motors
    servo*             p_servo_top;         ///< Pointer to the top
       servo
    servo*             p_servo_bottom;      ///< Pointer to the bottom
       servo


    unsigned char      finger_configuration[8];
    unsigned char      output[14];
    bool               flag_output_change;
    unsigned char      input_character;
    unsigned char      character_to_output;
    unsigned char      motor_to_stop;
    unsigned char      motor_to_start;
    unsigned char      motor_to_init;
    bool               flag_interference_thumb;
    bool               flag_interference_index;
    bool               flag_interference_middle;
    bool               flag_interference_ring;
    bool               flag_interference_pinky;
    bool               flag_motors_enabled;
    bool               flag_ready_to_output;
    bool               flag_stop_motors;
    bool               flag_start_motors;
    bool               flag_init_motors;
    unsigned char      character_step;
    unsigned char      i;

public:
    // The constructor creates a new task object
    task_output (task_timer&, time_stamp&, base_text_serial*,
       base_text_serial*, slave_picker*, servo*, servo*);

    // The run method is where the task actually performs its function
    char run (char);

    void set_new_character(unsigned char);

    void stop_motor (void);
    void start_motor (void);
    bool motors_enabled(void);
    bool query_motor (unsigned char);
    void init_motor (void);
    //void set_motor (unsigned char);
```

```
        void output_to_motor(unsigned char, unsigned char);
        bool ready_to_output(void);

        void open_thumb(void);
        void open_index(void);
        void open_middle(void);
        void open_ring(void);
        void open_pinky(void);

        void thumb_flat_up(void);
        void thumb_fold_up(void);
        void thumb_fold_in(void);
        void thumb_fold_out(void);
        void thumb_stretch(void);
        void thumb_curl(void);

        void index_stretch(void);
        void index_curl(void);
        void index_clench(void);
        void index_vert_clench(void);
        void index_cross(void);
        void index_u(void);
        void index_fold(void);

        void middle_stretch(void);
        void middle_curl(void);
        void middle_clench(void);
        void middle_vert_clench(void);
        void middle_fold(void);

        void ring_stretch(void);
        void ring_curl(void);
        void ring_clench(void);

        void pinky_stretch(void);
        void pinky_curl(void);
        void pinky_clench(void);

        void wrist_default(void);
        void wrist_bent(void);
        void wrist_bent_and_twisted(void);
        void wrist_twisted(void);
        void wrist_z1(void);
        void wrist_z2(void);
        void wrist_z3(void);
};

#endif
```

Code Block G.9: Output Task Class task_output.cpp

```
//*****************************************************************************
/** \file task_output.cpp
 *   This file contains a task class for running a user interface for the
    motor
 *   controller demonstration. It has a single-state task which just reads
    the
 *   serial port to see if the user typed anything, and acts if s/he has
    done so.
 *
 *  Revisions:
 *    \li 02-06-2008 JRR Original file
 *    \li 05-15-2008 JRR Modified to work with two motor drivers rather
    than one
 *    \li 03-08-2009 JRR Added code to test A/D converter
 *    \li 01-19-2011 JRR Updated calls to newer version of stl_task
    constructor
 *
 *  License:
 *   This file released under the Lesser GNU Public License, version 2.
    This program
 *   is intended for educational use only, but it is not limited thereto.
 */
//*****************************************************************************


#include <stdlib.h>
#include <avr/io.h>
#include "lib/rs232int.h"
#include "lib/stl_timer.h"
#include "lib/stl_task.h"
#include "servo.h"
#include "slave_picker.h"       // The class that sets the multiplexer pins
//#include "motor.h"
#include "task_output.h"
#include "lib/global_debug.h"

#define MOTOR_SWITCH_DDR DDRD
#define MOTOR_SWITCH_PORT PORTD
#define MOTOR_SWITCH_PIN PIND6


//---------------------------------------------------------------------------
/** This constructor creates a user interface task object. It checks if
   the user has
 *  typed a meaningful command at the serial port and if so tells the
    motor controller
 *  what to do.
 *  @param a_timer  A reference to the real-time measuring timer for tasks
 *  @param t_stamp  A timestamp which contains the time between runs of
    this task
```

```
 *  @param p_mo_task A pointer to a motor control task to be ordered around
 *  @param p_timer  A pointer to the main real-time clock object in use
 *  @param p_a_to_d A pointer to the A/D converter which measures voltages
 *  @param p_ser    A pointer to a serial device for sending and receiving
     messages
 */

task_output::task_output (task_timer& a_timer, time_stamp& t_stamp,
    base_text_serial* p_ser_comp, base_text_serial* p_ser_slave,
    slave_picker* p_slave_picker, servo* p_servotop, servo* p_servobottom)
    : stl_task (a_timer, t_stamp)
{

    // Assign pointers
    p_serial_comp = p_ser_comp;
    p_serial_slave = p_ser_slave;
    p_slave_chooser = p_slave_picker;
    //p_motors = p_the_motors;
    p_servo_top = p_servotop;
    p_servo_bottom = p_servobottom;

    for (i = 0; i < 8; i++)
    {
        finger_configuration[i] = 0;
    }

    for (i = 1; i < 14; i++)
    {
        output[i] = 0;
    }

    // Initialize variables
    flag_interference_thumb = false;
    flag_interference_index = false;
    flag_interference_middle = false;
    flag_interference_ring = false;
    flag_interference_pinky = false;
    flag_stop_motors = false;
    flag_start_motors = false;
    flag_init_motors = false;
    character_step = 1;
    motor_to_init = 1;
    motor_to_start = 1;
    motor_to_stop = 1;

    MOTOR_SWITCH_DDR |= (1 << MOTOR_SWITCH_PIN);
    MOTOR_SWITCH_PORT &= ~(1 << MOTOR_SWITCH_PIN);

    //*p_serial_comp << endl << "Output task initialized." << endl;
}
```

```
//-------------------------------------------------------------------------------------
/** This is the function which runs when it is called by the task
    scheduler. It causes
 *  the motor to move back and forth, having several states to cause such
     motion.
 *  @param state The state of the task when this run method begins running
 *  @return The state to which the task will transition, or
     STL_NO_TRANSITION if no
 *    transition is called for at this time
 */

char task_output::run (char state)
{
    //*p_serial_comp << endl << "Out State " << state << endl;

    switch(state)
    {
        // Wait for output change
        case(0):
            if (flag_stop_motors)
            {
                return(3);
            }
            else if (flag_start_motors)
            {
                return(5);
            }
            else if(flag_init_motors)
            {
                return(7);
            }
            else if(flag_output_change)
            {
                flag_output_change = false;
                return(1); // Go to state 1 (Check for interferences)
            }
            else
            {
                flag_ready_to_output = true;
                return(STL_NO_TRANSITION);
            }
            return(STL_NO_TRANSITION);
            break;
        // Check for interferences
        case(1):
            flag_ready_to_output = false;
            if(flag_interference_thumb)
            {
                open_thumb();
                flag_interference_thumb = false;
            }
```

```
    if(flag_interference_index)
    {
        open_index();
        flag_interference_index = false;
    }
    if(flag_interference_middle)
    {
        open_middle();
        flag_interference_middle = false;
    }
    if(flag_interference_ring)
    {
        open_ring();
        flag_interference_ring = false;
    }
    if(flag_interference_pinky)
    {
        open_pinky();
        flag_interference_pinky = false;
    }
    return(2);
    break;
// Process outputs
case(2):
    if (!flag_motors_enabled)
    {
        flag_motors_enabled = true;
    }
    // Parse character
    switch(character_to_output)
    {
        case('0'):
        case('O'):
        case('o'):
            thumb_curl();
            index_curl();
            middle_curl();
            ring_curl();
            pinky_curl();
            wrist_default();
            return(0);
            break;
        case('1'):
            thumb_flat_up();
            index_stretch();
            middle_clench();
            ring_clench();
            pinky_clench();
            wrist_default();
            return(0);
            break;
```

```
case('2'):
case('V'):
case('v'):
    thumb_flat_up();
    index_stretch();
    middle_stretch();
    ring_clench();
    pinky_clench();
    wrist_default();
    return(0);
    break;
case('3'):
    thumb_stretch();
    index_stretch();
    middle_stretch();
    ring_clench();
    pinky_clench();
    wrist_default();
    return(0);
    break;
case('4'):
case('b'):
case('B'):
    switch(character_step)
    {
        case(1):
            thumb_fold_out();
            index_stretch();
            middle_stretch();
            ring_stretch();
            pinky_stretch();
            wrist_default();
            character_step++;
            return(STL_NO_TRANSITION);
        case(2):
            thumb_fold_in();
            flag_interference_thumb = true;
            character_step = 1;
            return(0);
        default:
            *p_serial_comp << endl << "Error character " <<
                character_to_output << " step " <<
                character_step << endl;
    }
    break;
case('5'):
    thumb_stretch();
    index_stretch();
    middle_stretch();
    ring_stretch();
    pinky_stretch();
```

```
        wrist_default();
        return(0);
        break;
    case('6'):
    case('W'):
    case('w'):
        switch(character_step)
        {
            case(1):
                thumb_fold_out();
                index_stretch();
                middle_stretch();
                ring_stretch();
                pinky_clench();
                wrist_default();
                character_step++;
                return(STL_NO_TRANSITION);
            case(2):
                thumb_fold_in();
                flag_interference_thumb = true;
                character_step = 1;
                return(0);
            default:
                *p_serial_comp << endl << "Error character " <<
                    character_to_output << " step " <<
                    character_step << endl;
        }
        break;
    case('7'):
        switch(character_step)
        {
            case(1):
                thumb_fold_out();
                index_stretch();
                middle_stretch();
                ring_clench();
                pinky_stretch();
                wrist_default();
                character_step++;
                return(STL_NO_TRANSITION);
            case(2):
                thumb_fold_in();
                flag_interference_thumb = true;
                character_step = 1;
                return(0);
            default:
                *p_serial_comp << endl << "Error character " <<
                    character_to_output << " step " <<
                    character_step << endl;
        }
        break;
```

```
case('8'):
    switch(character_step)
    {
        case(1):
            thumb_fold_out();
            index_stretch();
            middle_clench();
            ring_stretch();
            pinky_stretch();
            wrist_default();
            character_step++;
            return(STL_NO_TRANSITION);
        case(2):
            thumb_fold_in();
            flag_interference_thumb = true;
            character_step = 1;
            return(0);
        default:
            *p_serial_comp << endl << "Error character " <<
                character_to_output << " step " <<
                character_step << endl;
    }
    break;
case('9'):
    thumb_flat_up();
    index_clench();
    middle_stretch();
    ring_stretch();
    pinky_stretch();
    wrist_default();
    return(0);
    break;
case('A'):
case('a'):
    thumb_flat_up();
    index_clench();
    middle_clench();
    ring_clench();
    pinky_clench();
    wrist_default();
    return(0);
    break;
case('C'):
case('c'):
    thumb_fold_out();
    index_curl();
    middle_curl();
    ring_curl();
    pinky_clench();
    wrist_default();
    return(0);
```

```
    break;
case('D'):
case('d'):
    thumb_curl();
    index_stretch();
    middle_curl();
    ring_curl();
    pinky_clench();
    wrist_default();
    return(0);
    break;
case('E'):
case('e'):
    switch(character_step)
    {
        case(1):
            thumb_fold_out();
            index_stretch();
            middle_stretch();
            ring_stretch();
            pinky_stretch();
            wrist_default();
            character_step++;
            return(STL_NO_TRANSITION);
        case(2):
            thumb_fold_in();
            index_curl();
            middle_curl();
            ring_curl();
            pinky_curl();
            flag_interference_thumb = true;
            flag_interference_index = true;
            flag_interference_middle = true;
            flag_interference_ring = true;
            flag_interference_pinky = true;
            character_step = 1;
            return(0);
        default:
            *p_serial_comp << endl << "Error character " <<
                character_to_output << " step " <<
                character_step << endl;
    }
    break;
case('F'):
case('f'):
    thumb_flat_up();
    index_clench();
    middle_stretch();
    ring_stretch();
    pinky_stretch();
    wrist_default();
```

```
    return(0);
    break;
case('G'):
case('g'):
    thumb_flat_up();
    index_stretch();
    middle_clench();
    ring_clench();
    pinky_clench();
    wrist_bent();
    return(0);
    break;
case('H'):
case('h'):
    thumb_flat_up();
    index_stretch();
    middle_stretch();
    ring_clench();
    pinky_clench();
    wrist_bent();
    return(0);
    break;
case('I'):
case('i'):
    thumb_flat_up();
    index_clench();
    middle_clench();
    ring_clench();
    pinky_stretch();
    wrist_default();
    return(0);
    break;
case('J'):
case('j'):
    switch(character_step)
    {
        case(1):
            thumb_flat_up();
            index_clench();
            middle_clench();
            ring_clench();
            pinky_stretch();
            wrist_default();
            character_step++;
            return(STL_NO_TRANSITION);
        case(2):
            wrist_bent();
            character_step++;
            return(STL_NO_TRANSITION);
        case(3):
            wrist_bent_and_twisted();
```

```
            character_step++;
            return(STL_NO_TRANSITION);
        case(4):
            wrist_twisted();
            character_step = 1;
            return(0);
    }
    break;
case('K'):
case('k'):
    switch(character_step)
    {
        case(1):
            thumb_flat_up();
            index_stretch();
            middle_stretch();
            ring_clench();
            pinky_clench();
            wrist_default();
            character_step++;
            return(STL_NO_TRANSITION);
        case(2):
        thumb_fold_up();
            flag_interference_thumb = true;
            character_step = 1;
            return(0);
        default:
            *p_serial_comp << endl << "Error character " <<
                character_to_output << " step " <<
                character_step << endl;
    }
    break;
case('L'):
case('l'):
    thumb_stretch();
    index_stretch();
    middle_clench();
    ring_clench();
    pinky_clench();
    wrist_default();
    return(0);
    break;
case('M'):
case('m'):
    switch(character_step)
    {
        case(1):
            thumb_fold_in();
            index_stretch();
            middle_stretch();
            ring_stretch();
```

```
                pinky_clench();
                wrist_default();
                character_step++;
                return(STL_NO_TRANSITION);
            case(2):
                index_vert_clench();
                middle_vert_clench();
                ring_curl();
                flag_interference_thumb = true;
                flag_interference_index = true;
                flag_interference_middle = true;
                flag_interference_ring = true;
                character_step = 1;
                return(0);
            default:
                *p_serial_comp << endl << "Error character " <<
                    character_to_output << " step " <<
                    character_step << endl;
        }
        break;
    case('N'):
    case('n'):
        switch(character_step)
        {
            case(1):
                thumb_fold_in();
                index_stretch();
                middle_stretch();
                ring_clench();
                pinky_clench();
                wrist_default();
                character_step++;
                return(STL_NO_TRANSITION);
            case(2):
                index_vert_clench();
                middle_vert_clench();
                flag_interference_thumb = true;
                flag_interference_index = true;
                flag_interference_middle = true;
                character_step = 1;
                return(0);
            default:
                *p_serial_comp << endl << "Error character " <<
                    character_to_output << " step " <<
                    character_step << endl;
        }
        break;
    case('P'):
    case('p'):
        thumb_fold_up();
        index_stretch();
```

```
        middle_fold();
        ring_clench();
        pinky_clench();
        wrist_bent();
        flag_interference_thumb = true;
        flag_interference_middle = true;
        return(0);
        break;
    case('Q'):
    case('q'):
        thumb_fold_out();
        index_fold();
        middle_clench();
        ring_clench();
        pinky_clench();
        wrist_bent();
        return(0);
        break;
    case('R'):
    case('r'):
        thumb_flat_up();
        index_cross();
        middle_clench();
        ring_clench();
        pinky_clench();
        wrist_default();
        flag_interference_index = true;
        return(0);
        break;
    case('S'):
    case('s'):
        switch(character_step)
        {
            case(1):
                thumb_fold_out();
                index_clench();
                middle_clench();
                ring_clench();
                pinky_clench();
                wrist_default();
                character_step++;
                return(STL_NO_TRANSITION);
            case(2):
                thumb_fold_in();
                flag_interference_thumb = true;
                character_step = 1;
                return(0);
            default:
                *p_serial_comp << endl << "Error character " <<
                    character_to_output << " step " <<
                    character_step << endl;
```

```
    }
    break;
case('T'):
case('t'):
    switch(character_step)
    {
        case(1):
            thumb_flat_up();
            index_vert_clench();
            middle_clench();
            ring_clench();
            pinky_clench();
            wrist_default();
            character_step++;
            return(STL_NO_TRANSITION);
        case(2):
            thumb_fold_in();
            flag_interference_thumb = true;
            flag_interference_index = true;
            character_step = 1;
            return(0);
        default:
            *p_serial_comp << endl << "Error character " <<
                character_to_output << " step " <<
                character_step << endl;
    }
    break;
case('U'):
case('u'):
    switch(character_step)
    {
        case(1):
            thumb_flat_up();
            index_stretch();
            middle_stretch();
            ring_clench();
            pinky_clench();
            wrist_default();
            character_step++;
            return(STL_NO_TRANSITION);
        case(2):
            index_u();
            flag_interference_index = true;
            character_step = 1;
            return(0);
        default:
            *p_serial_comp << endl << "Error character " <<
                character_to_output << " step " <<
                character_step << endl;
    }
    break;
```

```
case('X'):
case('x'):
    switch(character_step)
    {
        case(1):
            thumb_fold_out();
            index_stretch();
            middle_clench();
            ring_clench();
            pinky_clench();
            wrist_default();
            character_step++;
            return(STL_NO_TRANSITION);
        case(2):
            thumb_fold_in();
            index_vert_clench();
            flag_interference_thumb = true;
            character_step = 1;
            return(0);
        default:
            *p_serial_comp << endl << "Error character " <<
                character_to_output << " step " <<
                character_step << endl;
    }
case('Y'):
case('y'):
    thumb_stretch();
    index_clench();
    middle_clench();
    ring_clench();
    pinky_stretch();
    wrist_default();
    break;
case('Z'):
case('z'):
    switch(character_step)
    {
        case(1):
            thumb_flat_up();
            index_clench();
            middle_clench();
            ring_clench();
            pinky_stretch();
            wrist_z1();
            character_step++;
            return(STL_NO_TRANSITION);
        case(2):
            wrist_z2();
            character_step++;
            return(STL_NO_TRANSITION);
        case(3):
```

```
                    wrist_z3();
                    character_step++;
                    return(STL_NO_TRANSITION);
                case(4):
                    wrist_bent();
                    character_step = 1;
                    return(0);
                default:
                    *p_serial_comp << endl << "Error character " <<
                        character_to_output << " step " <<
                        character_step << endl;
            }
        }

        return(0); // Go to state 2 (output) when done
        break;
    // Send Stop Command to a Motor
    case(3):
        flag_stop_motors = false;
        p_slave_chooser->choose(motor_to_stop);
        if(p_serial_slave->ready_to_send())
        {
            *p_serial_slave << "S";
        }
        return(4);
        break;
    // Wait for confirmation on stop command
    case(4):
        if(p_serial_slave->check_for_char())
        {
            input_character = p_serial_slave->getchar();
            if (input_character == 's')
                *p_serial_comp << endl << "Motor " << motor_to_stop << "
                    stopped";
            else
                *p_serial_comp << endl << "Motor stop error " <<
                    motor_to_stop << endl;

            if (motor_to_stop == 10)
            {
                motor_to_stop = 1;
                return(0);
            }
            else
            {
                motor_to_stop++;
                return(3);
            }
        }
        else
        {
```

```
        return(STL_NO_TRANSITION);
    }
// Send Start Command to a Motor
case(5):
    flag_start_motors = false;
    p_slave_chooser->choose(motor_to_start);
    if(p_serial_slave->ready_to_send())
    {
        *p_serial_slave << "G";
    }
    return(6);
    break;
// Wait for confirmation on start command
case(6):
    if(p_serial_slave->check_for_char())
    {
        input_character = p_serial_slave->getchar();
        if (input_character == 'g')
            *p_serial_comp << endl << "Motor " << motor_to_start <<
                " enabled";
        else
            *p_serial_comp << endl << "Motor start error " <<
                motor_to_start << endl;

        if (motor_to_start == 10)
        {
            motor_to_start = 1;
            return(0);
        }
        else
        {
            motor_to_start++;
            return(5);
        }
    }
    else
    {
        return(STL_NO_TRANSITION);
    }
// Send initialization command to one motor
case(7):
    p_slave_chooser->choose(motor_to_init);
    if(p_serial_slave->ready_to_send())
    {
        if (motor_to_init > 0 && motor_to_init < 10)
            character_to_output = motor_to_init + 0x30;
        else if (motor_to_init == 10)
            character_to_output = '0';
        else
            *p_serial_comp << endl << "Motor conf error " <<
                motor_to_init << endl;
```

```
                *p_serial_slave << character_to_output;
                return(8);
            }
            break;
        // Wait for initialization response from motor
        case(8):
            if(p_serial_slave->check_for_char())
            {
                input_character = p_serial_slave->getchar();
                if (input_character == '!')
                    *p_serial_comp << endl << "Motor " << motor_to_init << "
                        initialized";
                else
                    *p_serial_comp << endl << "Motor init error " <<
                        motor_to_init << endl;

                if (motor_to_init == 10)
                {
                    motor_to_init = 1;
                    return(0);
                }
                else
                {
                    motor_to_init++;
                    return(7);
                }
            }
            else
            {
                return(STL_NO_TRANSITION);
            }
            break;
        default:
            return(0);
            break;

    }
    // If we get here, no transition is called for
    return (STL_NO_TRANSITION);
}

void task_output::set_new_character(unsigned char outchar)
{
    character_to_output = outchar;
    *p_serial_comp << endl << "New output character: " << ascii <<
        character_to_output << numeric << endl;
    flag_output_change = true;
}

void task_output::stop_motor(void)
```

```
{
    motor_to_stop = 1;
    flag_stop_motors = true;
    flag_motors_enabled = false;
}

void task_output::start_motor(void)
{
    motor_to_start = 1;
    flag_start_motors = true;
    flag_motors_enabled = true;
}

bool task_output::motors_enabled(void)
{
    return(flag_motors_enabled);
}

bool task_output::query_motor(unsigned char motornum)
{
    p_slave_chooser->choose(motornum);
    if(p_serial_slave->ready_to_send())
    {
        *p_serial_slave << "Q";
        input_character = p_serial_comp->getchar();  // Wait for response
        if (input_character == 'Q')
        {
            return(false);
        }
        else if (input_character == 'q')
        {
            return(true);
        }
        else
        *p_serial_comp << endl << "Motor query error " << motornum << endl;
        return(false);
    }
    else
    {
        *p_serial_comp << endl << "Serial port not ready to send to motor "
            << motornum << endl;
        return(false);
    }

}

void task_output::init_motor(void)
{
    motor_to_init = 1;
    flag_init_motors = true;
}
```

```
void task_output::output_to_motor (unsigned char motornumber, unsigned
    char output_value)
{
    //*p_serial_comp << "Select motor " << numeric << motornumber << endl;

    if (motornumber <= 10 && motornumber >= 0)
    {
        p_slave_chooser->choose(motornumber);
        *p_serial_slave << output_value;
        *p_serial_comp << ascii << output_value << numeric;
    }
    else if (motornumber == 11)
    {
        if(output_value == 1)
        {
            MOTOR_SWITCH_PORT |= (1 << MOTOR_SWITCH_PIN);
        }
        else if(output_value == 0)
        {
            MOTOR_SWITCH_PORT &= ~(1 << MOTOR_SWITCH_PIN);
        }
    }
    else if (motornumber == 12)
    {
        p_servo_top->output(output_value);
    }
    else if (motornumber == 13)
    {
        p_servo_bottom->output(output_value);
    }
    else
    {
        *p_serial_comp << "Motor number outside bounds" << endl;
    }
}

bool task_output::ready_to_output(void)
{
    return (flag_ready_to_output);
}

void task_output::open_thumb(void)
{
    *p_serial_comp << endl << "thumb" << endl;
    output_to_motor(5,'a');
}

void task_output::open_index(void)
{
    *p_serial_comp << endl << "index" << endl;
```

```
    output_to_motor(1,'a');
    output_to_motor(11,0);
}

void task_output::open_middle(void)
{
    *p_serial_comp << endl << "middle" << endl;
    output_to_motor(2,'a');
}

void task_output::open_ring(void)
{
    *p_serial_comp << endl << "ring" << endl;
    output_to_motor(3,'a');
}

void task_output::open_pinky(void)
{
    *p_serial_comp << endl << "pinky" << endl;
    output_to_motor(4,'a');
}

void task_output::thumb_flat_up(void)
{
    *p_serial_comp << endl << "thumb" << endl;
    output_to_motor(5,'a');
    output_to_motor(6,'a');
    output_to_motor(7,'a');
    output_to_motor(8,'a');
}

void task_output::thumb_fold_up(void)
{
    *p_serial_comp << endl << "thumb" << endl;
    output_to_motor(5,'e');
    output_to_motor(6,'a');
    output_to_motor(7,'a');
    output_to_motor(8,'a');
}

void task_output::thumb_fold_in(void)
{
    *p_serial_comp << endl << "thumb" << endl;
    output_to_motor(5,'c');
    output_to_motor(6,'c');
    output_to_motor(7,'e');
    output_to_motor(8,'a');
}

void task_output::thumb_fold_out(void)
{
```

```
    *p_serial_comp << endl << "thumb" << endl;
    output_to_motor(5,'e');
    output_to_motor(6,'a');
    output_to_motor(7,'b');
    output_to_motor(8,'b');
}

void task_output::thumb_stretch(void)
{
    *p_serial_comp << endl << "thumb" << endl;
    output_to_motor(5,'a');
    output_to_motor(6,'e');
    output_to_motor(7,'a');
    output_to_motor(8,'a');
}

void task_output::thumb_curl(void)
{
    *p_serial_comp << endl << "thumb" << endl;
    output_to_motor(5,'e');
    output_to_motor(6,'b');
    output_to_motor(7,'b');
    output_to_motor(8,'b');
}

void task_output::index_stretch(void)
{
    *p_serial_comp << endl << "index" << endl;
    output_to_motor(1,'a');
    output_to_motor(9,'a');
}

void task_output::index_curl(void)
{
    *p_serial_comp << endl << "index" << endl;
    output_to_motor(1,'c');
    output_to_motor(9,'c');
}

void task_output::index_clench(void)
{
    *p_serial_comp << endl << "index" << endl;
    output_to_motor(1,'e');
    output_to_motor(9,'e');
}

void task_output::index_vert_clench(void)
{
    *p_serial_comp << endl << "index" << endl;
    output_to_motor(1,'a');
    output_to_motor(9,'e');
```

```
}

void task_output::index_cross(void)
{
    *p_serial_comp << endl << "index" << endl;
    output_to_motor(1,'c');
    output_to_motor(9,'a');
    output_to_motor(11,1);
}

void task_output::index_u(void)
{
    *p_serial_comp << endl << "index" << endl;
    output_to_motor(1,'a');
    output_to_motor(9,'a');
    output_to_motor(11,1);
}

void task_output::index_fold(void)
{
    *p_serial_comp << endl << "index" << endl;
    output_to_motor(1,'e');
    output_to_motor(9,'a');
}

void task_output::middle_stretch(void)
{
    *p_serial_comp << endl << "middle" << endl;
    output_to_motor(2,'a');
    output_to_motor(10,'a');
}

void task_output::middle_curl(void)
{
    *p_serial_comp << endl << "middle" << endl;
    output_to_motor(2,'c');
    output_to_motor(10,'c');
}

void task_output::middle_clench(void)
{
    *p_serial_comp << endl << "middle" << endl;
    output_to_motor(2,'e');
    output_to_motor(10,'e');
}

void task_output::middle_vert_clench(void)
{
    *p_serial_comp << endl << "middle" << endl;
    output_to_motor(2,'a');
    output_to_motor(10,'e');
```

```
}

void task_output::middle_fold(void)
{
    *p_serial_comp << endl << "middle" << endl;
    output_to_motor(2,'e');
    output_to_motor(10,'a');
}

void task_output::ring_stretch(void)
{
    *p_serial_comp << endl << "ring" << endl;
    output_to_motor(3,'a');
}

void task_output::ring_curl(void)
{
    *p_serial_comp << endl << "ring" << endl;
    output_to_motor(3,'c');
}

void task_output::ring_clench(void)
{
    *p_serial_comp << endl << "ring" << endl;
    output_to_motor(3,'e');
}

void task_output::pinky_stretch(void)
{
    *p_serial_comp << endl << "pinky" << endl;
    output_to_motor(4,'a');
}

void task_output::pinky_curl(void)
{
    *p_serial_comp << endl << "pinky" << endl;
    output_to_motor(4,'c');
}

void task_output::pinky_clench(void)
{
    *p_serial_comp << endl << "pinky" << endl;
    output_to_motor(4,'e');
}

void task_output::wrist_default(void)
{
    *p_serial_comp << endl << "wrist" << endl;
    output_to_motor(12,0);
    output_to_motor(13,0);
}
```

```
void task_output::wrist_bent(void)
{
    *p_serial_comp << endl << "wrist" << endl;
    output_to_motor(12,90);
    output_to_motor(13,0);
}

void task_output::wrist_bent_and_twisted(void)
{
    *p_serial_comp << endl << "wrist" << endl;
    output_to_motor(12,90);
    output_to_motor(13,90);
}

void task_output::wrist_twisted(void)
{
    *p_serial_comp << endl << "wrist" << endl;
    output_to_motor(12,0);
    output_to_motor(13,90);
}

void task_output::wrist_z1(void)
{
    *p_serial_comp << endl << "wrist" << endl;
    output_to_motor(12,45);
    output_to_motor(13,45);
}

void task_output::wrist_z2(void)
{
    *p_serial_comp << endl << "wrist" << endl;
    output_to_motor(12,45);
    output_to_motor(13,0);
}

void task_output::wrist_z3(void)
{
    *p_serial_comp << endl << "wrist" << endl;
    output_to_motor(12,90);
    output_to_motor(13,45);
}
```

Code Block G.10: User Interface Task Header task_user.h

```
//*****************************************************************************
/** \file task_user.h
 *   This file contains a task class for running a user interface for the
    motor
 *   controller demonstration. It has a single-state task which just reads
    the
 *   serial port to see if the user typed anything, and acts if s/he has
    done so.
 *
 *  Revisions:
 *    \li 02-06-2008 JRR Original file
 *    \li 05-15-2008 JRR Modified to work with two motor drivers rather
    than one
 *    \li 03-08-2009 JRR Added code to test A/D converter
 *    \li 01-19-2011 JRR Updated calls to newer version of stl_task
    constructor
 *
 *  License:
 *  This file released under the Lesser GNU Public License, version 2.
    This program
 *  is intended for educational use only, but it is not limited thereto.
 */
//*****************************************************************************


#include "lib/stl_timer.h"

#ifndef _TASK_USER_H_
#define _TASK_USER_H_

#define MAX_SENTENCE_SIZE 255

//---------------------------------------------------------------------------
/** This class contains a task which moves a motorized lever back and
   forth.
 *  WARNING: This task uses an older version of parent class stl_task, and
    its
 *           constructor parameters are out of date. Use it as an example,
    but
 *           do not attempt to just copy the parameters.
 */

class task_user : public stl_task
{
   protected:

      base_text_serial*  p_serial_comp;      ///< Pointer to serial
         device for computer
      base_text_serial*  p_serial_slave;     ///< Pointer to serial
         device for slave
```

```
    slave_picker*     p_slave_chooser;      ///< Pointer to slave
        picker for mux pins
    character_database* p_character_database; ///< Pointer to the
        character database
    task_output*      p_task_output;        ///< Pointer to the output
        task

    unsigned char     input_character;      ///< Input character from
        serial device
    bool              flag_message_printed; ///< Boolean to prevent
        messages from being printed repeatedly
    char              backspace;            ///< Backspace variable
    unsigned char     index;                ///< Index for character
        array
    unsigned char     character_to_output;  ///< Placeholder for
        character to output
    unsigned char     character_to_test;    ///< Placeholder for
        character to test
    unsigned char     steps;                ///< Number of steps in a
        character
    unsigned char     current_step;         ///< Current gesture output
        step
    unsigned char     output_delay;         ///< Number of times to
        skip displaying a character
    unsigned char     current_delay;        ///< Remaining number of
        delay counts
    unsigned char     output_configuration; ///< Finger configuration
        to output to output task
    unsigned char     encoder_reading;      ///< Encoder reading
        retrieved from motor

    bool              flag_period;          ///< Flag to indicate a
        period character
    bool              flag_comma;           ///< Flag to indicate a
        comma character
    bool              flag_space;           ///< Flag to indicate a
        space character
    bool              flag_delay_countdown; ///< Flag to indicate the
        output delay countdown has begun
    bool              flag_outputting_letter; ///< Flag to indicate
        whether outputting a letter or pause

    queue<char, unsigned char, MAX_SENTENCE_SIZE> character_buffer;
        ///< Character buffer

    unsigned char     i_motor;


public:
    // The constructor creates a new task object
    task_user (task_timer&, time_stamp&, base_text_serial*,
```

```
            base_text_serial*, slave_picker*, task_output*);

        // The run method is where the task actually performs its function
        char run (char);

};

#endif
```

Code Block G.11: User Interface Task Class task_user.cpp

```
//****************************************************************************
/** \file task_user.cpp
 *    This file contains a task class for running a user interface for the
    motor
 *    controller demonstration. It has a single-state task which just reads
    the
 *    serial port to see if the user typed anything, and acts if s/he has
    done so.
 *
 *  Revisions:
 *    \li 02-06-2008 JRR Original file
 *    \li 05-15-2008 JRR Modified to work with two motor drivers rather
    than one
 *    \li 03-08-2009 JRR Added code to test A/D converter
 *    \li 01-19-2011 JRR Updated calls to newer version of stl_task
    constructor
 *
 *  License:
 *    This file released under the Lesser GNU Public License, version 2.
    This program
 *    is intended for educational use only, but it is not limited thereto.
 */
//****************************************************************************


#include <stdlib.h>
#include <avr/io.h>
#include "lib/rs232int.h"
#include "lib/stl_timer.h"
#include "lib/stl_task.h"
#include "slave_picker.h"       // The class that sets the multiplexer pins
#include "lib/queue.h"
#include "character.h"          // The class that stores character info
#include "character_database.h"
#include "task_output.h"
#include "task_user.h"
#include "lib/global_debug.h"


//----------------------------------------------------------------------------
/** This constructor creates a user interface task object. It checks if
   the user has
 *  typed a meaningful command at the serial port and if so tells the
    motor controller
 *  what to do.
 *  @param a_timer  A reference to the real-time measuring timer for tasks
 *  @param t_stamp  A timestamp which contains the time between runs of
    this task
 *  @param p_mo_task A pointer to a motor control task to be ordered around
 *  @param p_timer  A pointer to the main real-time clock object in use
```

```
 *  @param p_a_to_d A pointer to the A/D converter which measures voltages
 *  @param p_ser    A pointer to a serial device for sending and receiving
     messages
 */

task_user::task_user (task_timer& a_timer, time_stamp& t_stamp,
    base_text_serial* p_ser_comp,
                    base_text_serial* p_ser_slave, slave_picker*
                        p_slave_picker,
                    task_output* p_output_task )
    : stl_task (a_timer, t_stamp)
{
    flag_message_printed = false; // Clear message_printed flag

    // Assign pointers
    p_serial_comp = p_ser_comp;
    p_serial_slave = p_ser_slave;
    p_slave_chooser = p_slave_picker;
    //p_character_database = p_char_dbase;
    p_task_output = p_output_task;

    character_buffer.flush(); // Flush character buffer

    backspace = 0x08;         // Backspace character for printing

    *p_serial_comp << endl << "User task initialized" << endl;

}

//-------------------------------------------------------------------------------------
/** This is the function which runs when it is called by the task
    scheduler. It causes
 *  the motor to move back and forth, having several states to cause such
     motion.
 *  @param state The state of the task when this run method begins running
 *  @return The state to which the task will transition, or
     STL_NO_TRANSITION if no
 *    transition is called for at this time
 */

char task_user::run (char state)
{
    //*p_serial_comp << endl << "Use State " << state << endl;
    switch(state)
    {
        // Home screen
        case(0):
            if(!flag_message_printed)
            {
                *p_serial_comp << endl << endl << "Robotic Fingerspelling
                    Hand" << endl << endl;
```

```
            *p_serial_comp << endl << "ESC Stop Motors" <<
                              endl << "C  Calibrate" <<
                              endl << "ENT Enter Sentence" <<
                              endl << "E  Encoder Query" <<
                              endl << "M  Manual Mode" << endl ;
            flag_message_printed = true;
        }
        if(p_serial_comp->check_for_char())
        {
            flag_message_printed = false;
            input_character = p_serial_comp->getchar();
            switch(input_character)
            {
                case(0x1B):    // Escape
                    return(1); // Go to state 1 (stop motors)
                    break;
                case('c'):     // Capital or lowercase C entered
                case('C'):
                    return(2); // Go to state 2 (calibration menu)
                    break;
                case(0x0D):    // Enter
                    return(4); // Go to state 4 (enter sentence)
                    break;
                case('E'):
                case('e'):
                    return(10); // Go to state 10 (Encoder query)
                    break;
                case('M'):
                case('m'):
                    return(13); // Go to state 13 (Manual mode)
                    break;
                default:
                    *p_serial_comp << endl << "Invalid command" << endl;
                    break;
            }
        }
        //*p_serial_comp << endl << "User task state 0" << endl;
        return(STL_NO_TRANSITION);
        break;
    // Stop motors
    case(1):
        *p_serial_comp << endl << "Sending stop command" << endl;
        p_task_output->stop_motor();
        return(0); // return to state 0 (home screen)
        break;
    // Print calibration messages
    case(2):
        *p_serial_comp << endl << "Calibrate which motor?" << endl <<
            endl << "1 - M1" << endl << "2 - M2"
        << endl << "3 - M3" << endl << "4 - M4" << endl << "5 - M5" <<
            endl << "6 - M6" << endl << "7 - M7"
```

```
    << endl << "8 - M8" << endl << "9 - M9" << endl << "0 - M10" <<
        endl << "ESC Cancel" << endl;
    return(3); // Process input in state 3
    break;
// Perform calibration
case(3):
    if(p_serial_comp->check_for_char())
    {
        input_character = p_serial_comp->getchar();  // Collect
            character

        // Clear character buffer
        while(p_serial_slave ->check_for_char())
        {
            p_serial_slave->getchar();
        }

        if( (input_character >= 0x31) && (input_character <= 0x39) )
        {
            // Subtract 0x30 from input character to get decimal
                value
            input_character = input_character - 0x30;

            // Set multiplexer to the proper pin
            p_slave_chooser->choose(input_character);

            // Output C character to clear encoder count in slave
            if(p_serial_slave->ready_to_send())
            {
                *p_serial_slave << 'C';
            }
            return(16);    // Wait for response in state 16
        }
        else if ( input_character == '0' )
        {
            // Choose multiplexer channel 10
            p_slave_chooser->choose(10);

            // Output C character to clear encoder count in slave
            if(p_serial_slave->ready_to_send())
            {
                *p_serial_slave << 'C';
            }
            return(16);    // Wait for response in state 16
        }
        else if (input_character == 0x1B) // Escape
        {
            return(0);     // Go back to state 0 (home screen)
        }
        else
        {
```

```
            *p_serial_comp << endl << "Invalid character" << endl;
            return(2); // Reprint message and remain in loop until
                valid character received
        }
    }
    break;
// Collect characters
case(4):
    // Print instructions and flush the character buffer the first
        time through.
    if(!flag_message_printed)
    {
        *p_serial_comp << endl << "Input sentence. Letters, numbers,
            commas, periods, and spaces only. 255 characters max."
        << endl << "Enter when done. Escape to quit." << endl << ">
            ";
        flag_message_printed = true;
        if(!character_buffer.is_empty()) // If character buffer is
            NOT empty
        {
            character_buffer.flush();    // flush character buffer
        }
    }

    // Collect characters
    if(p_serial_comp->check_for_char())  // Check if character
        received
    {
        input_character = p_serial_comp->getchar();  // collect
            character if so.

        // Process character

        // Is it a number (between hex 30 and 39), capital letter
            (between hex 41 and 5A), space, comma, or period?
        // If so it can be stored directly
        if( ( (input_character >= '0')&&(input_character <= '9')
            )||( (input_character >= 'A')&&(input_character <= 'Z' )
            )||(input_character == ' ')||(input_character ==
            ',')||(input_character == '.') )
        {
            if (character_buffer.num_items() <= MAX_SENTENCE_SIZE )
                // If it is and there's room in the character buffer
            {
                *p_serial_comp << ascii << input_character <<
                    numeric; // Echo character to the screen
                character_buffer.put(input_character);            //
                    store it directly into the character buffer
            }
            else                                                  //
                If the character buffer is full, scream at the user.
```

```
    {
        *p_serial_comp << endl << "TOO MANY CHARACTERS" <<
            endl;
    }
}
// If not a number or capital letter, is it a lowercase
    letter (between hex 61 and 7A)?
else if ( (input_character >= 'a')&&(input_character <= 'z')
    )
{
    if (character_buffer.num_items() <= MAX_SENTENCE_SIZE )
        // If it is and there's room in the character buffer
    {

        input_character = input_character - ('a' - 'A');  //
            convert it to a capital letter first.
        character_buffer.put(input_character);            //
            Store it in the character buffer
        *p_serial_comp << ascii << input_character <<
            numeric; // Echo character to the screen
    }
    else                                                  //
        If the character buffer is full, scream at the user.
    {
        *p_serial_comp << endl << "TOO MANY CHARACTERS" <<
            endl;
    }
}
// If backspace (hex 08)
else if (input_character == 0x08)
{
    *p_serial_comp << ascii << backspace << " " << backspace
        << numeric; // Backspace, space, backspace to step
        back, erase, and move the cursor back.
    character_buffer.delete_one();          // Delete the
        last character stored in the buffer.
}
// If question mark or exclamation point, store as period.
else if ( (input_character == '?')||(input_character == '!')
    )
{
    *p_serial_comp << ascii << input_character << numeric;
        // Echo character to the screen
    input_character = '.';                              //
        convert it to a period first
    character_buffer.put(input_character);              //
        Store it in the character buffer
}
// If Enter is pressed, user is done.
else if (input_character == 0x0D)
{
```

```
            *p_serial_comp << endl << "Parsing sentence." << endl;
            flag_message_printed = false;
            return(5); // Go to state 5
        }
        // If ESC is pressed, user is quitting
        else if (input_character == 0x1B)
        {
            *p_serial_comp << endl << "Quitting" << endl;
            flag_message_printed = false;
            return(0); // Go to state 5
        }
        // If any other characters are pressed
        else
        {
            // Don't echo or store anything
            return(STL_NO_TRANSITION);
        }

    }

    return(STL_NO_TRANSITION); // Don't leave the state until Enter
        is pressed.
    break;
// Prepare one character for output
case(5):
    if(character_buffer.num_items()) // If character buffer is not
        empty
    {
        character_to_output = character_buffer.get();
               // Retrieve the character

        // If it's not a pause character, collect information.
        if ((character_to_output != '.')||(character_to_output !=
            ',')||(character_to_output != ' '))
        {
            flag_outputting_letter = true;
        }
        // If it's a pause character, set the proper flag.
        else
        {
            flag_outputting_letter = false;
            switch(character_to_output)
            {
                case('.'):
                    flag_period = true;
                    break;
                case(','):
                    flag_comma = true;
                    break;
                case(' '):
                    flag_space = true;
```

```
                break;
            default:
                break;
        }
    }
    return(6); // Now go to state 6 to figure out timing
}
else
{
    return(9); // Printing is done. Go to state 9 to print the
        end message.
}
break;
// Timing calculations
case(6):
    if(flag_comma || flag_space || flag_period )
    {
        if(flag_comma)
        {
            output_delay = 60;
        }
        else if(flag_space)
        {
            output_delay = 40;
        }
        else
        {
            output_delay = 80;
        }
    }
    else
    {
        output_delay = 20;
    }
    current_delay = 0;
    current_step = 0;
    return(7); // Go to state 7 to start outputting
    break;
// Delay
case(7):
    if(current_delay == output_delay) // Wait until delay counter
        increments up to desired value
    {
        current_delay = 0;       // Clear delay count
        return(8);               // Now go to state 8 to output data
    }
    else
    {
        current_delay++;
        return(STL_NO_TRANSITION); // Keep delaying until ready
    }
```

```
    break;
// Output values
case(8):
    // Enable motors if they're disabled
    if (!(p_task_output -> motors_enabled()))
    {
        p_task_output -> init_motor();
        p_task_output -> start_motor();
    }

    // Output the character
    if (p_task_output -> ready_to_output())
    {
        p_task_output -> set_new_character(character_to_output);
        return(5); // Output next letter
    }
    else
    {
        return(STL_NO_TRANSITION); // Wait
    }
    break;
// Done
case(9):
    if (p_task_output -> ready_to_output() &&
        flag_outputting_letter == true)
    {
        *p_serial_comp << endl << "Message done. Returning to
            message prompt." << endl;
        flag_outputting_letter = false;
        return(4); // Return to message prompt
    }
    else
    {
        return(STL_NO_TRANSITION); // Wait
    }
    break;
// Encoder Reading Prompt
case(10):
    *p_serial_comp << endl << "Read which encoder?" << endl << endl
        << "1 - M1" << endl << "2 - M2"
    << endl << "3 - M3" << endl << "4 - M4" << endl << "5 - M5" <<
        endl << "6 - M6" << endl << "7 - M7"
    << endl << "8 - M8" << endl << "9 - M9" << endl << "0 - M10" <<
        endl << "ESC Cancel" << endl;
    return(11);
    break;
// Encoder Reading Processing
case(11):
    if(p_serial_comp->check_for_char())
    {
        input_character = p_serial_comp->getchar();  // Collect
```

```
            character

        // Clear slave character buffer
        while(p_serial_slave ->check_for_char())
        {
            p_serial_slave->getchar();
        }

        if( (input_character >= 0x31) && (input_character <= 0x39) )
        {
            // Subtract 0x30 from input character to get decimal
                value
            input_character = input_character - 0x30;

            // Set multiplexer to the proper pin
            p_slave_chooser->choose(input_character);

            // Output E character to trigger encoder count return
            if(p_serial_slave->ready_to_send())
            {
                *p_serial_slave << 'E';
            }
            return(12);    // Go back to state 10 (encoder prompt)
        }
        else if ( input_character == '0' )
        {
            // Choose multiplexer channel 10
            p_slave_chooser->choose(10);

            // Output E character to trigger encoder count return
            if(p_serial_slave->ready_to_send())
            {
                *p_serial_slave << 'E';
            }
            return(12);    // Go back to state 10 (encoder prompt)
        }
        else if (input_character == 0x1B) // Escape
        {
            return(0);     // Go back to state 0 (home screen)
        }
        else
        {
            *p_serial_comp << endl << "Invalid character" << endl;
            return(10);    // Go back to state 10 (encoder prompt)
        }
    }
    break;
// Wait for encoder count return
case(12):
    if(p_serial_slave->check_for_char())
    {
```

```
            encoder_reading = p_serial_slave -> getchar();
            encoder_reading = encoder_reading*4; // Convert from 8 bit
                truncated count to 10 bit count
            *p_serial_comp << endl << "Encoder reading: " << numeric <<
                encoder_reading << endl;
            return(10); // Return to encoder prompt
        }
        else if (p_serial_comp ->check_for_char())
        {
            input_character = p_serial_comp->getchar();
            if (input_character == 0x1B)
            {
                *p_serial_comp << endl << "Reading cancelled" << endl;
                return(10);
            }
        }
        else
        {
            return(STL_NO_TRANSITION);
        }
        break;
    // Manual Mode Prompt
    case(13):
        *p_serial_comp << endl << "Control which motor?" << endl <<
            endl << "1 - M1" << endl << "2 - M2"
        << endl << "3 - M3" << endl << "4 - M4" << endl << "5 - M5" <<
            endl << "6 - M6" << endl << "7 - M7"
        << endl << "8 - M8" << endl << "9 - M9" << endl << "0 - M10" <<
            endl << "ESC Cancel" << endl;
        return(14);
        break;
    // Manual Mode Processing
    case(14):
        if(p_serial_comp->check_for_char())
        {
            input_character = p_serial_comp -> getchar();  // Collect
                character

            // Clear slave character buffer
            while(p_serial_slave ->check_for_char())
            {
                p_serial_slave->getchar();
            }

            if( (input_character >= 0x31) && (input_character <= 0x39) )
            {
                // Subtract 0x30 from input character to get decimal
                    value
                input_character = input_character - 0x30;

                // Set multiplexer to the proper pin
```

```
            p_slave_chooser->choose(input_character);
        }
        else if ( input_character == '0' )
        {
            // Choose multiplexer channel 10
            p_slave_chooser->choose(10);
        }
        else if (input_character == 0x1B) // Escape
        {
            return(0);     // Go back to state 0 (home screen)
        }
        else
        {
            *p_serial_comp << endl << "Invalid character" << endl;
            return(13);    // Go back to state 13 (encoder prompt)
        }
        *p_serial_comp << endl << "Input command. ESC to exit." <<
            endl;
        return(15);
    }
    break;
case(15):
    // Command Prompt
    if(p_serial_comp -> check_for_char())
    {
        input_character = p_serial_comp -> getchar();

        if (input_character != 0x1B)
        {
            if(p_serial_slave -> ready_to_send())
            {
                *p_serial_slave << input_character;
                *p_serial_comp << endl << "Sent " << ascii <<
                    input_character << numeric << " to motor." <<
                    endl;
            }
        }
        else
        {
            return(13); // Return to motor list if ESC pressed
        }
        return(STL_NO_TRANSITION);   // Stay here unless ESC pressed
    }
    return(STL_NO_TRANSITION);
    break;
// Wait for calibration response
case(16):
    if (p_serial_slave -> check_for_char())
    {
        input_character = p_serial_slave -> getchar();
        if (input_character == 'c')
```

```
                {
                    *p_serial_comp << endl << "Calibration successful." <<
                        endl;
                }
                else
                {
                    *p_serial_comp << endl << "Calibration failed." << endl;
                }
                return(2);
            }
            else if (p_serial_comp -> check_for_char())
            {
                input_character = p_serial_comp -> getchar();
                if (input_character == 0x1B)
                {
                    *p_serial_comp << endl << "Calibration cancelled" <<
                        endl;
                    return(2);
                }
            }
            else
            {
                return (STL_NO_TRANSITION);
            }
            break;
        default:
            break;
    }
    // If we get here, no transition is called for
    return (STL_NO_TRANSITION);
};
```

Code Block G.12: Slave Picker Header slave_picker.h

```
//*****************************************************************************
/** \file slave_picker.h
 *   This file contains a task class for running a user interface for the
    motor
 *   controller demonstration. It has a single-state task which just reads
    the
 *   serial port to see if the user typed anything, and acts if s/he has
    done so.
 *
 *  Revisions:
 *    \li 02-06-2008 JRR Original file
 *    \li 05-15-2008 JRR Modified to work with two motor drivers rather
    than one
 *    \li 03-08-2009 JRR Added code to test A/D converter
 *    \li 01-19-2011 JRR Updated calls to newer version of stl_task
    constructor
 *
 *  License:
 *  This file released under the Lesser GNU Public License, version 2.
    This program
 *  is intended for educational use only, but it is not limited thereto.
 */
//*****************************************************************************

#include "lib/base_text_serial.h"

#ifndef _SLAVE_PICKER_H_
#define _SLAVE_PICKER_H_

//---------------------------------------------------------------------------
/** This class contains a task which moves a motorized lever back and
   forth.
 *  WARNING: This task uses an older version of parent class stl_task, and
    its
 *           constructor parameters are out of date. Use it as an example,
    but
 *           do not attempt to just copy the parameters.
 */

class slave_picker
{
    protected:
        unsigned char     pinarray[4];  // Create pin array
        base_text_serial*  p_serial_comp;       ///< Pointer to serial
            device for computer

    public:
        // The constructor creates a new task object
        slave_picker (base_text_serial*);
```

```
        // The run method is where the task actually performs its function
        void choose (unsigned char);
};

#endif
```

Code Block G.13: Slave Picker Class slave_picker.cpp

```
//******************************************************************************
/** \file slave_picker.cpp
 *   This file contains a task class for running a user interface for the
    motor
 *   controller demonstration. It has a single-state task which just reads
    the
 *   serial port to see if the user typed anything, and acts if s/he has
    done so.
 *
 *  Revisions:
 *    \li 02-06-2008 JRR Original file
 *    \li 05-15-2008 JRR Modified to work with two motor drivers rather
    than one
 *    \li 03-08-2009 JRR Added code to test A/D converter
 *    \li 01-19-2011 JRR Updated calls to newer version of stl_task
    constructor
 *
 *  License:
 *   This file released under the Lesser GNU Public License, version 2.
    This program
 *   is intended for educational use only, but it is not limited thereto.
 */
//******************************************************************************

#include <stdlib.h>
#include <avr/io.h>
#include "slave_picker.h"
#include "lib/global_debug.h"

//------------------------------------------------------------------------------
/** This constructor creates a slave_picker object. It outputs the correct
   pins to
 *  choose multiplexer outputs to make sure the master is communicating
    with the right
 *  slave chips.
 */

slave_picker::slave_picker (base_text_serial* p_ser_comp)
{
    // Assign pointers
    p_serial_comp = p_ser_comp;

    // Clear array
    for(unsigned char i = 0; i < 4; i++)
    {
        pinarray[i] = 0;
    }

    DDRA = 0xFF;   // Set all pins in register A to be outputs
}
```

```
//--------------------------------------------------------------------------------
/** This is the function which runs when it is called by the task
    scheduler. It causes
 *  the motor to move back and forth, having several states to cause such
     motion.
 *  @param pinnumber The output pin on the multiplexers
 *  @return Nothing
 */

void slave_picker::choose (unsigned char pinnumber)
{
    *p_serial_comp << numeric << " Mot " << pinnumber;

    // Split number into individual bits
    for(unsigned char i = 0; i < 4; i++)
    {
        pinarray[i] = (0b00000001 << i) & pinnumber;
    }

    // Output to pins
    if(pinarray[0])
    {
        PORTA |= (1 << PINA0);
        PORTA |= (1 << PINA4);
    }
    else
    {
        PORTA &= ~(1 << PINA0);
        PORTA &= ~(1 << PINA4);
    }

    if(pinarray[1])
    {
        PORTA |= (1 << PINA1);
        PORTA |= (1 << PINA5);
    }
    else
    {
        PORTA &= ~(1 << PINA1);
        PORTA &= ~(1 << PINA5);
    }

    if(pinarray[2])
    {
        PORTA |= (1 << PINA2);
        PORTA |= (1 << PINA6);
    }
    else
    {
```

```
        PORTA &= ~(1 << PINA2);
        PORTA &= ~(1 << PINA6);
    }

    if(pinarray[3])
    {
        PORTA |= (1 << PINA3);
        PORTA |= (1 << PINA7);
    }
    else
    {
        PORTA &= ~(1 << PINA3);
        PORTA &= ~(1 << PINA7);
    }

}
```

Code Block G.14: Servo Header servo.h

```
//*****************************************************************************
/** \file servo.h
 *    This file contains a class which runs the servo.
 *
 *  Revisions:
 *    \li 02-24-2012 JV Created file
 *
 *  License:
 *    This file released under the Lesser GNU Public License, version 2.
    This program
 *    is intended for educational use only, but its use is not limited
    thereto.
 */
//*****************************************************************************

#ifndef _SERVO_H_
#define _SERVO_H_                    ///< Prevents multiple inclusion of
    file

#define SERVO_DDR DDRD
#define SERVO_PORT PORTD
#define SERVO_PIN1 PIND5
#define SERVO_PIN2 PIND4


//-----------------------------------------------------------------------------
/** This class operates two servos using Timer Counter 1
 */

class servo
{
    protected:
        unsigned char which_motor;       ///< Is this object for motor 1 or
            2?

    public:
        /// This constructor creates a motor controller object.
        servo (unsigned char);

        /// This method sets the OCR value to output to the servo
        void output (unsigned char);
};

#endif // _SERVO_H_
```

Code Block G.15: Servo Class servo.cpp

```
//*****************************************************************************
/** \file servo.cpp
 *   This file contains a class which runs the motor driver on an ME405
    board,
 *   version 0.60+. This version has two VNH3SP30 motor drivers controlled
    by the
 *   ATmega128 microcontroller. Please see the class definition for class
    motor405
 *   for a list of which pins are connected to which signals.
 *
 *  Revisions:
 *   \li 02-24-2012 JV Created file
 *
 *  License:
 *   This file released under the Lesser GNU Public License, version 2.
    This program
 *   is intended for educational use only, but its use is not limited
    thereto.
 */
//*****************************************************************************

#include <avr/io.h>
#include "servo.h"


//-----------------------------------------------------------------------------
/** This constructor enables the servo driver
 *  @param servo_number The number of the motor to be controlled, motor 1
    or motor 2
 */

servo::servo (unsigned char servo_number)
{
    static bool timer_set_up = false;   // Set up timer only on first call

    which_motor = servo_number;         // Save the number of the servo to
        control

    // If nobody has yet set up the timer which both motors share, set it
        up now
    if (!timer_set_up)
    {
        // Set up Timer 1 in 16-bit fast PWM mode with the
        //prescaler at I/O clock / 8, with the PWM in normal polarity
        TCCR1A = (1 << WGM11);
        TCCR1B = (1 << CS11) | (1 << WGM12) | (1 << WGM13);
        ICR1 = 49999;

        // The block above should only be called once when the first PWM is
            set up
```

```
        timer_set_up = true;
    }

    // If motor 1 is being set up, use Port C for mode and OC1B = Port B
        pin 6 for PWM
    if (which_motor == 1)
    {
        // Set output compare for fast PWM, normal polarity
        TCCR1A |= (1 << COM1A1);

        // Set up the servo pin as an output
        SERVO_DDR |= (1 << SERVO_PIN1);

        // Set the duty cycle to zero, also for safety
        OCR1AH = 0;
        OCR1AL = 0;
    }
    // If motor 2 is being set up, use Port D for mode and OC1A = Port B
        pin 5 for PWM
    else if (which_motor == 2)
    {
        // Set output compare for fast PWM, normal polarity
        TCCR1A |= (1 << COM1B1);

        // Set up the servo pin as an output
        SERVO_DDR |= (1 << SERVO_PIN2);

        // Set the duty cycle to zero, also for safety
        OCR1BH = 0;
        OCR1BL = 0;
    }
    else
    {
        return;
        // An invalid motor number has been chosen -- we can't do anything
    }
}

//-------------------------------------------------------------------------------------
/** This method outputs a servo signal to the servo based on an input angle
 *  @param angle The angle the servo should turn
 */

void servo::output (unsigned char angle)
{
    unsigned short int new_ocr = (unsigned short int) ((unsigned long int)
        2500 * angle / 180) + 2500;

    if (which_motor == 1)
    {
        OCR1A = new_ocr;
```

```
    }
    else if (which_motor == 2)
    {
        OCR1B = new_ocr;
    }
}
```

G.3 Master Libraries from ME 405 Mechatronics

Code Block G.16: State Transition Logic Timer Header stl_timer.h

```
//*****************************************************************************
/** \file stl_timer.h
 *    This file contains a class which runs a task timer whose resolution
    is one
 *    microsecond. The timer is used to help schedule the execution of
    tasks' run()
 *    methods, and it is also used to keep track of real time in general.
    The run()
 *    methods can be run from a main loop by checking periodically if a
    task time
 *    has expired, or they can be called directly from a timer interrupt
    service
 *    routine.
 *
 *  Revisions:
 *    \li 08-07-2007 JRR Created this file as daytimer.* with 1 second
    interrupts
 *    \li 08-08-2007 JRR Added event triggers
 *    \li 12-23-2007 JRR Made more general by allowing faster interrupt
    rates
 *    \li 01-05-2008 JRR Converted from time-of-day version to microsecond
    version
 *    \li 03-27-2008 JRR Added operators + and - for time stamps
 *    \li 03-31-2008 JRR Merged in stl_us_timer (int, long) and set_time
    (int, long)
 *    \li 05-15-2008 JRR Changed to use Timer 3 so Timer 1 can run motor
    PWM's
 *    \li 05-31-2008 JRR Changed time calculations to use CPU_FREQ_MHz from
    Makefile
 *    \li 01-04-2009 JRR Now uses CPU_FREQ_Hz (rather than MHz) for better
    precision
 *    \li 11-24-2009 JRR Changed CPU_FREQ_Hz to F_CPU to match AVR-LibC's
    name
 *
 *  License:
 *    This file copyright 2007 by JR Ridgely. It is released under the
    Lesser GNU
 *    public license, version 2.
 */
//*****************************************************************************

/// These defines prevent this file from being included more than once in
    a *.cc file
#ifndef _STL_TIMER_H_
#define _STL_TIMER_H_
```

```
#define F_CPU 20000000
// Check that the user has set the CPU frequency in the Makefile; if not,
    complain
#ifndef F_CPU
    #error The macro F_CPU must be set in the Makefile.
#endif

// If Timer 3 exists (as on the ATmega128), we use it for the microsecond
    timer rather
// than Timer 1, as Timer 1 is used for the motor PWM's on the ME405 board
#ifdef TCNT3
    #define TMR_TCNT_REG  TCNT3          ///< Register that holds the time
        count
    #define TMR_intr_vect TIMER3_OVF_vect ///< The timer overflow
        interrupt vector
#else
    #define TMR_TCNT_REG  TCNT1          ///< Register that holds the time
        count
    #define TMR_intr_vect TIMER1_OVF_vect ///< The timer overflow
        interrupt vector
#endif // __AVR_ATmega128__


//---------------------------------------------------------------------------
/** This union holds a 32-bit time count. The count can be accessed as a
    single 32-bit
 *  number, as two 16-bit integers placed together, or as an array of
     8-bit characters.
 *  This is helpful because the time measurement we use consists of the
     number in a
 *  16-bit hardware counter and a 16-bit overflow count which should be
     put together
 *  into one time number. The characters are handy for looking at
     individual bits.
 */

typedef union
{
    uint32_t whole;                     ///< All the data as one 32-bit
        number
    uint16_t half[2];                   ///< The data as an array of
        16-bit ints
    uint8_t quarters[4];                ///< The data as an array of 8-bit
        chars
} time_data_32;


//---------------------------------------------------------------------------
/** This class holds a time stamp which is used to measure the passage of
    real time in
```

```
 *  the world around an AVR processor. This version of the time stamp
    implements a
 *  32-bit time counter that runs at one megahertz. There's a 16-bit
    number which is
 *  copied directly from a 16-bit hardware counter and another 16-bit
    number which is
 *  incremented every time the hardware counter overflows; the combination
    of the two
 *  is a 32-bit time measurement. The time stamp is to be interpreted and
    printed as
 *  containing a whole number of seconds and a number of microseconds. The
    conversion
 *  between seconds and microseconds is determined by the CPU clock
    frequency, which
 *  is specified in the macro F_CPU.
 */

class time_stamp
{
    protected:
        /// This union holds the time stamp's data and allows it to be
            accessed as two
        /// 16-bit halves or one 32-bit whole as required by the application
        time_data_32 data;

    public:
        /// This constructor creates an empty time stamp
        time_stamp (void);

        /// This constructor creates a time stamp and initializes all its
            data
        time_stamp (uint32_t);

        /// This constructor creates a time stamp with the given seconds
            and microsec.
        time_stamp (uint16_t, uint32_t);

        /// This method fills the timestamp with the given value
        void set_time (uint32_t);

        /// This method fills the timestamp with the given seconds and
            microseconds
        void set_time (uint16_t, uint32_t);

        /// This method reads out all the timestamp's data as one 32-bit
            number
        uint32_t get_raw_time (void);

        /// This method returns the number of seconds in the time stamp
        uint16_t get_seconds (void);
```

```
        /// This method returns the number of microseconds in the time stamp
        uint32_t get_microsec (void);

        /// This overloaded addition operator adds two time stamps together
        time_stamp operator + (const time_stamp&);

        /// This overloaded subtraction operator finds the time between two
            time stamps
        time_stamp operator - (const time_stamp&);

        /// This overloaded addition operator adds two time stamps together
        void operator += (const time_stamp&);

        /// This overloaded subtraction operator finds the time between two
            time stamps
        void operator -= (const time_stamp&);

        /// This overloaded operator divides a time stamp by the given
            divisor
        void operator /= (const uint32_t&);

        /// This overloaded equality operator tests if all time fields are
            the same
        bool operator == (const time_stamp&);

        /// This operator tests if a time stamp is greater (later) or equal
            to this one
        bool operator >= (const time_stamp&);

        /// This overloaded operator tests if a time stamp is greater
            (later) than this
        bool operator > (const time_stamp&);

        /// This operator tests if a time stamp is less (earlier) than this
            one
        bool operator <= (const time_stamp&);

        /// This operator tests if a time stamp is less (earlier) or equal
            to this one
        bool operator < (const time_stamp&);

        // This declaration gives permission for objects of class
            task_timer to access
        // the private and/or protected data belonging to objects of this
            class
        friend class task_timer;
};


//-------------------------------------------------------------------------------
```

```
/** This class implements a timer to synchronize the operation of tasks on
    an AVR. The
 *  timer is implemented as a combination of a 16-bit hardware timer
     (Timer 1 is the
 *  usual choice) and a 16-bit overflow counter. The two timers' data is
     combined to
 *  produce a 32-bit time count which is used to decide when tasks run.
     WARNING: This
 *  timer does not keep track of the time of day, and it overflows after a
     little more
 *  than an hour of use. Another version of the stl_timer can be used when
     longer time
 *  periods need to be kept track of, to lower precision.
 */

class task_timer
{
    protected:
        /// This time stamp object holds a most recently measured time
        time_stamp now_time;

    public:
        task_timer (void);              /// Constructor creates an empty
            timer
        void save_time_stamp (time_stamp&); /// Save current time in a
            timestamp
        time_stamp& get_time_now (void); /// Get the current time

        /// This method sets the current time to the time in the given time
            stamp
        bool set_time (time_stamp&);
};

//-------------------------------------------------------------------------------
// These operators allow timestamps to be written to serial ports 'cout'
    style
base_text_serial& operator<< (base_text_serial&, time_stamp&);
base_text_serial& operator<< (base_text_serial&, task_timer&);

#endif // _STL_TIMER_H_
```

Code Block G.17: State Transition Logic Timer Class stl_timer.cpp

```
//*********************************************************************************
/** \file stl_timer.cpp
 *    This file contains a class which runs a task timer whose resolution
    is one
 *    microsecond. The timer is used to help schedule the execution of
    tasks' run()
 *    functions. The functions can be run from a main loop by checking
    periodically
 *    if a task time has expired, or they can be called directly from the
    timer
 *    interrupt service routine in this file, or they can be called from
    some other
 *    hardware interrupt service routine; in the last case this file isn't
    involved.
 *
 *  Revisions:
 *    \li 08-07-2007 JRR Created this file as daytimer.* with 1 second
    interrupts
 *    \li 08-08-2007 JRR Added event triggers
 *    \li 12-23-2007 JRR Made more general by allowing faster interrupt
    rates
 *    \li 01-05-2008 JRR Converted from time-of-day version to microsecond
    version
 *    \li 03-27-2008 JRR Added operators + and - for time stamps
 *    \li 03-31-2008 JRR Merged in stl_us_timer (int, long) and set_time
    (int, long)
 *    \li 05-15-2008 JRR Changed to use Timer 3 so Timer 1 can run motor
    PWM's
 *    \li 05-31-2008 JRR Changed time calculations to use CPU_FREQ_MHz from
    Makefile
 *    \li 01-04-2009 JRR Now uses CPU_FREQ_Hz (rather than MHz) for better
    precision
 *    \li 11-24-2009 JRR Changed CPU_FREQ_Hz to F_CPU to match AVR-LibC's
    name
 *
 *  License:
 *    This file copyright 2007 by JR Ridgely. It is released under the
    Lesser GNU
 *    public license, version 2.
 */
//*********************************************************************************

#include <stdlib.h>                     // Used for itoa()
#include <string.h>                     // Header for character string
    functions
#include <avr/interrupt.h>              // For using interrupt service
    routines

#include "base_text_serial.h"           // Base for text-type serial port
    objects
```

```
#include "stl_timer.h"                  // Header for this file


//-----------------------------------------------------------------------------
// This variables is only used to allow the interrupt service routine to
    update the
// measured time whenever a timer interrupt occurs, and allow a timer
    object to read
// the time. The user should not have any reason to read or write it.

/** This variable holds the number of times the hardware timer has
    overflowed. This
 *  number is equivalent to the upper 16 bits of a 32-bit timer, and is so
     used. */

uint16_t ust_overflows = 0;


//-----------------------------------------------------------------------------
/** This constructor creates a time stamp object, with the time set to
    zero.
 */

time_stamp::time_stamp (void)
{
    set_time (0L);
}


//-----------------------------------------------------------------------------
/** This constructor creates a time stamp object and fills the time
    stamp's variables
 *  with the given values.
 *  @param a_time A 32-bit time number with which the time stamp is filled
 */

time_stamp::time_stamp (uint32_t a_time)
{
    set_time (a_time);
}


//-----------------------------------------------------------------------------
/** This constructor creates a time stamp object and fills the time
    stamp's variables
 *  with the number of seconds and microseconds given.
 *  @param sec A 16-bit number of seconds to preload into the time stamp
 *  @param microsec A 32-bit number of microseconds to preload into the
     time stamp
 */
```

```
time_stamp::time_stamp (uint16_t sec, uint32_t microsec)
{
    set_time (sec, microsec);
}


//-----------------------------------------------------------------------------
/** This method fills the time stamp with the given value.
 *  @param a_time A 32-bit time number with which the time stamp is filled
 */

void time_stamp::set_time (uint32_t a_time)
{
    data.whole = a_time;
}


//-----------------------------------------------------------------------------
/** This method fills the time stamp with the given numbers of seconds and
    microseconds.
 *  @param sec A 16-bit number of seconds to preload into the time stamp
 *  @param microsec A 32-bit number of microseconds to preload into the
     time stamp
 */

void time_stamp::set_time (uint16_t sec, uint32_t microsec)
{
    data.whole = (microsec * (F_CPU / 1000000UL)) / 8UL;
    data.whole += (uint16_t)sec * (F_CPU / 8UL);
}


//-----------------------------------------------------------------------------
/** This method allows one to get the time reading from this time stamp as
    a long
 *  integer containing the number of time ticks.
 *  @return The time stamp's raw data
 */

uint32_t time_stamp::get_raw_time (void)
{
    return (data.whole);
}


//-----------------------------------------------------------------------------
/** This method returns the number of seconds in the time stamp.
 *  @return The number of whole seconds in the time stamp
 */

uint16_t time_stamp::get_seconds (void)
```

```
{
    return ((uint16_t)(data.whole / (F_CPU / 8)));
}


//-------------------------------------------------------------------------------------
/** This method returns the number of microseconds in the time stamp,
    after the seconds
 *  are subtracted out.
 *  @return The number of microseconds, that is, the fractional part of
     the time stamp
 */

uint32_t time_stamp::get_microsec (void)
{
    return ((data.whole % (F_CPU / 8UL)) * 8 / (F_CPU / 1000000UL));
}


//-------------------------------------------------------------------------------------
/** This overloaded addition operator adds another time stamp's time to
    this one. It
 *  can be used to find the time in the future at which some event is to
     be caused to
 *  happen, such as the next time a task is supposed to run.
 *  @param addend The other time stamp which is to be added to this one
 *  @return The newly created time stamp
 */

time_stamp time_stamp::operator + (const time_stamp& addend)
{
    time_stamp ret_stamp;
    ret_stamp.data.whole = this->data.whole + addend.data.whole;

    return ret_stamp;
}


//-------------------------------------------------------------------------------------
/** This overloaded subtraction operator finds the duration between this
    time stamp's
 *  recorded time and a previous one.
 *  @param previous An earlier time stamp to be compared to the current one
 *  @return The newly created time stamp
 */

time_stamp time_stamp::operator - (const time_stamp& previous)
{
    time_stamp ret_stamp;
    ret_stamp.data.whole = this->data.whole - previous.data.whole;
```

```
    return ret_stamp;
}


//-------------------------------------------------------------------------------
/** This overloaded addition operator adds another time stamp's time to
    this one. It
 *  can be used to find the time in the future at which some event is to
     be caused to
 *  happen, such as the next time a task is supposed to run.
 *  @param addend The other time stamp which is to be added to this one
 */

void time_stamp::operator += (const time_stamp& addend)
{
    data.whole += addend.data.whole;
}


//-------------------------------------------------------------------------------
/** This overloaded subtraction operator finds the duration between this
    time stamp's
 *  recorded time and a previous one. Note that the data in this timestamp
     is
 *  replaced with that duration.
 *  @param previous An earlier time stamp to be compared to the current one
 */

void time_stamp::operator -= (const time_stamp& previous)
{
    data.whole -= previous.data.whole;
}


//-------------------------------------------------------------------------------
/** This overloaded division operator divides a time stamp by the given
    integer. Note
 *  that the data in this timestamp is replaced by the quotient.
 *  @param divisor The number by which the time stamp is to be divided
 */

void time_stamp::operator /= (const uint32_t& divisor)
{
    data.whole /= divisor;
}


//-------------------------------------------------------------------------------
/** This overloaded equality test operator checks if all the time
    measurements in some
 *  other time stamp are equal to those in this one.
```

```
 *  @param other A time stamp to be compared to this one
 *  @return True if the time stamps contain equal data, false if they don't
 */

bool time_stamp::operator == (const time_stamp& other)
{
    if (other.data.whole == data.whole)
    {
        return (true);
    }
    else
    {
        return (false);
    }
}


//---------------------------------------------------------------------------
/** This overloaded inequality operator checks if this time stamp is
    greater than or
 *  equal to another. The method used to check greater-than-ness needs to
     work across
 *  timer overflows, so the following technique is used: subtract the
     other time stamp
 *  from this one as unsigned 32-bit numbers, then check if the result is
     positive (in
 *  which case this time is greater) or not.
 *  @param other A time stamp to be compared to this one
 *  @return True if this time stamp is greater than or equal to the other
     one
 */

bool time_stamp::operator >= (const time_stamp& other)
{
    int32_t difference = (int32_t)(data.whole - other.data.whole);
    if (difference >= 0L)
    {
        return (true);
    }
    else
    {
        return (false);
    }
}


//---------------------------------------------------------------------------
/** This operator tests if a time stamp is greater than this one. It
    subtracts the
 *  other time stamp from this time stamp, then checks if the result is
     positive.
```

```
 *  @param other A time stamp to be compared to this one
 *  @return True if this time stamp is greater than the other one
 */

bool time_stamp::operator > (const time_stamp& other)
{
    int32_t difference = (int32_t)(data.whole - other.data.whole);
    if (difference > 0L)
    {
        return (true);
    }
    else
    {
        return (false);
    }
}


//-----------------------------------------------------------------------------
/** This operator tests if a time stamp is less than or equal to this one.
    It subtracts
 *  the other time stamp from this time stamp, then checks if the result
     is not
 *  positive.
 *  @param other A time stamp to be compared to this one
 *  @return True if this time stamp is less than or equal to the other one
 */

bool time_stamp::operator <= (const time_stamp& other)
{
    int32_t difference = (int32_t)(data.whole - other.data.whole);
    if (difference <= 0L)
    {
        return (true);
    }
    else
    {
        return (false);
    }
}


//-----------------------------------------------------------------------------
/** This operator tests if a time stamp is greater than this one. It
    subtracts the
 *  other time stamp from this time stamp, then checks if the result is
     negative.
 *  @param other A time stamp to be compared to this one
 *  @return True if this time stamp is less than the other one
 */
```

```
bool time_stamp::operator < (const time_stamp& other)
{
    int32_t difference = (int32_t)(data.whole - other.data.whole);
    if (difference < 0L)
    {
        return (true);
    }
    else
    {
        return (false);
    }
}



//-------------------------------------------------------------------------------
/** This constructor creates a daytime task timer object. It sets up the
    hardware timer
 *  to count at ~1 MHz and interrupt on overflow. Note that this method
     does not enable
 *  interrupts globally, so the user must call sei() at some point to
     enable the timer
 *  interrupts to run and time to actually be measured.
 */

task_timer::task_timer (void)
{
    #ifdef TCNT3                        // If Timer 3 exists, we'll use it
        TCCR3A = 0x00;                  // Set to normal 16-bit counting
            mode
        TCCR3B = (1 << CS31);           // Set prescaler to main clock / 8
        #ifdef ETIMSK                   // For ATmega128
            ETIMSK |= (1 << TOIE3);     // Set Timer 3 overflow interrupt
                enable
        #endif
        #ifdef TIMSK3                   // For ATmega1281
            TIMSK3 |= (1 << TOIE3);     // Set Timer 3 overflow interrupt
                enable
        #endif
    #else                               // No Timer 3, so use Timer 1
        TCCR1A = 0x00;                  // Set to normal 16-bit counting
            mode
        TCCR1B = (1 << CS11);           // Set prescaler to main clock / 8
        #ifdef TIMSK1                   // If there's a TIMSK1, set it
            TIMSK1 |= (1 << TOIE1);     // Enable Timer 1 overflow
                interrupt
        #else                           // If no TIMSK1, there's a TIMSK
            TIMSK |= (1 << TOIE1);      // Enable Timer 1 overflow
                interrupt
        #endif
    #endif
```

```
}


//---------------------------------------------------------------------------
/** This method grabs the current time stamp from the hardware and
    overflow counters.
 *  In order to prevent the data changing during the time when it's being
     read (which
 *  would cause invalid data to be saved), interrupts are disabled while
     the time data
 *  is being copied.
 *  @param the_stamp Reference to a time stamp variable which will hold
     the time
 */

void task_timer::save_time_stamp (time_stamp& the_stamp)
{
    uint8_t temp_sreg = SREG;           // Store interrupt flag status
    the_stamp.data.half[0] = TMR_TCNT_REG; // Get hardware count
    cli ();                             // Prevent interruption
    the_stamp.data.half[1] = ust_overflows; // Get overflow counter data
    SREG = temp_sreg;                   // Re-enable interrupts if they
        were on
}


//---------------------------------------------------------------------------
/** This method saves the current time in the internal time stamp
    belonging to this
 *  object, then returns a reference to the time stamp so that the caller
     can use it as
 *  a measurement of what the time is now.
 */

time_stamp& task_timer::get_time_now (void)
{
    uint8_t temp_sreg = SREG;           // Store interrupt flag status
    now_time.data.half[0] = TMR_TCNT_REG; // Get hardware count
    cli ();                             // Prevent interruption
    now_time.data.half[1] = ust_overflows; // Get overflow counter data
    SREG = temp_sreg;                   // Re-enable interrupts if they
        were on

    return (now_time);                  // Return a reference to the
        current time
}


//---------------------------------------------------------------------------
/** This method sets the timer to a given value. It's not likely that this
    method will
```

```
 *  be used, but it is provided for compatibility with other task timer
    implementations
 *  that measure times of day (in hours, minutes, and seconds) and do need
    to be set by
 *  user programs.
 *  @param t_stamp A reference to a time stamp containing the time to be
    set
 */

bool task_timer::set_time (time_stamp& t_stamp)
{
    uint8_t temp_sreg = SREG;           // Store interrupt flag status
    cli ();                             // Prevent interruption
    ust_overflows = t_stamp.data.half[1];
    TMR_TCNT_REG = t_stamp.data.half[0];
    SREG = temp_sreg;                   // Re-enable interrupts if they
       were on
}


//-----------------------------------------------------------------------------
/** This overloaded operator allows a time stamp to be printed on a serial
   device such
 *  as a regular serial port or radio module in text mode. This allows
    lines to be set
 *  up in the style of 'cout.' The timestamp is always printed as a
    decimal number.
 *  @param serial A reference to the serial-type object to which to print
 *  @param stamp A reference to the time stamp to be displayed
 */

base_text_serial& operator<< (base_text_serial& serial, time_stamp& stamp)
{
    // Get the time in seconds and microseconds
    uint16_t seconds = stamp.get_seconds ();
    uint32_t microseconds = stamp.get_microsec ();

    serial << seconds;
    serial.putchar ('.');

    // For the digits in the fractional part, write 'em in backwards
       order. We can't
    // use itoa here because we need leading zeros
    for (uint32_t divisor = 100000; divisor > 0; divisor /= 10)
    {
        serial.putchar (microseconds / divisor + '0');
        microseconds %= divisor;
    }

    return (serial);
}
```

```
//---------------------------------------------------------------------------
/** This overloaded operator allows the task timer to print the current
    time on a serial
 *  device such as a regular serial port or radio module in text mode.
     This allows lines
 *  to be set up in the style of 'cout.' The printing is done by the
     time_stamp class;
 *  this method just calls the timer's get_time_now() method to get a time
     stamp, then
 *  has the time stamp print itself.
 *  @param serial A reference to the serial-type object to which to print
 *  @param tmr A reference to the timer whose time is to be displayed
 */

base_text_serial& operator<< (base_text_serial& serial, task_timer& tmr)
{
    serial << tmr.get_time_now ();
    return (serial);
}


//---------------------------------------------------------------------------
/** This is the interrupt service routine which is called whenever there
    is a compare
 *  match on the 16-bit timer's counter. Nearly all AVR processors have a
     16-bit timer
 *  called Timer 1, and this is the one we use here.
 */

ISR (TMR_intr_vect)
{
    ust_overflows++;
}
```

Code Block G.18: State Transition Logic Task Header stl_task.h

```
//****************************************************************************
/** \file stl_task.h
 *  This file contains the definition of a task class. This class
    implements the
 *  tasks in a multitasking system. These tasks are intended to be run in a
 *  cooperative multitasking framework, but they can also be worked into an
 *  interrupt framework or run within an RTOS.
 *
 *  Revisions
 *    \li 04-21-07 JRR Original of this file, derived from UCB's TranRun4
    and
 *                     simplified greatly for efficient use in AVR processors
 *    \li 05-07-07 JRR Small bug fixes
 *    \li 06-01-08 JRR Changed debugging/trace to take advantage of
    base_text_serial
 *    \li 06-03-08 JRR Cleaned up comments, got rid of Doxygen warnings
 *
 *  License:
 *    This file released under the Lesser GNU Public License, version 2.
    This program
 *    is intended for educational use only, but it is not limited thereto.
 */
//****************************************************************************

/// This define prevents this .h file from being included more than once
   in a .cc file
#ifndef _STL_TASK_H_
#define _STL_TASK_H_

#include "stl_timer.h"                  // Include the header for the task
   timer


//----------------------------------------------------------------------------
/** This define specifies a no-transition code which means that the next
   state will be
 *  the same as the current state, and with no self-transition.
 */

const char STL_NO_TRANSITION = 0xFF;


//----------------------------------------------------------------------------
/** This enumeration lists the possible operational states of a task.
   These states are
 *  not the same as the states which are programmed by the user; these
    states are only
 *  used by the task scheduler to figure out which task to run at a given
    time
 *  according to priorities (if used) and which tasks are ready to be run.
```

```
 */

enum task_op_state
{
    TASK_RUNNING,         ///< The task's run() function is executing
    TASK_PENDING,         ///< The task needs to run again as soon as
       possible
    TASK_WAITING,         ///< The task is waiting for its next run time to
       occur
    TASK_BLOCKED,         ///< The task cannot run because a resource is
       unavailable
    TASK_SUSPENDED        ///< The task is turned off until its resume() is
       called
};


//---------------------------------------------------------------------------
/** This class implements the behavior of a task in the context of a
   multitasking
 *  system. Each task runs "simultaneously" with other tasks. This means,
    of course,
 *  that execution switches quickly from one task to another quite quickly
    under the
 *  control of the main execution scheduler. Since a task can be in any of
    several
 *  states at a given time, a task object will have a list of state
    methods which
 *  can be run and be able to switch between those states.
 *
 *  \section task_usage Usage
 *   The programmer uses this class by deriving a descendent class from
    it. Then
 *   s/he creates a run() method in the new class which implements the
    states.
 *   In the run() method, there's a big switch statement which chooses
    which state
 *   code is to run when the run() method is called. Checking the time and
    deciding
 *   when to run is taken care of by the schedule() method, which only
    runs the
 *   task's state code when the task's time interval has been reached; if
    the time
 *   interval is set to 0, the state code is run every time run() is
    called. State
 *   transitions are detected and tracked by this class for testing and
    debugging
 *   purposes.
 *
 *  \section task_options Options
 *   \li Serial port debugging can be enabled by defining SERIAL_DEBUG and
    calling
```

```
*       the constructor which is given a pointer to a uart object. The
    task will
*       then write information about state transitions to the serial port
    during
*       operation. This option is intended to be used during debugging,
    then turned
*       off for production code, as serial device writing takes up time
    and memory,
*       and of course it requires a serial device to be present and
    connected.
*    \li Execution time profiling can be enabled by defining STL_PROFILE.
    This
*       option causes the execution times of the state functions to be
    measured
*       and a simple set of performance data to be kept. Performance data
    can be
*       written to a serial port at a convenient time, generally after
    the system
*       has been run in test for a while.
*
*  \section task_intrn Internal Organization
*    At any time, a task is in both a <i>user state</i> and an
    <i>operational
*    state</i>. The user state is the state which the user manipulates; it
    chooses
*    the action that the run() method will take each time it is called by
    the
*    scheduler. The user state is the state which appears in the user's
    state
*    transition diagrams. The operational state shows how the scheduler is
    treating a
*    task - whether the task is running, waiting for its time to run,
    ready to run,
*    suspended, etc. Operational states for all tasks follow the "state
    diagram"
*    below.
*
*  \dot
*  digraph op_states {
*    node [ shape = ellipse, fontname = Arial, fontsize = 10];
*      waiting  [ label = "Waiting " ];
*      suspended [ label = "Suspended "];
*      pending  [ label = "Pending " ];
*      running  [ label = "Running " ];
*      start    [ label = "", shape = "plaintext" ];
*    edge [ arrowhead = "normal", style = "solid", fontname = Courier,
    fontsize = 10 ];
*      start->waiting [ label = " start " ];
*      waiting->running [ label = " run() " ];
*      running->waiting [ label = " run() finished " ];
*      waiting->suspended [ label = "suspend() " ];
```

```
 *     suspended->waiting [ label = "resume() " ];
 *     running->pending [ label = "run_again_ASAP() " ];
 *     pending->running [ label = " run() " ];
 *  }
 *  \enddot
 *    Blocked tasks (due to resource contention) are not currently
    implemented.
 *
 *    When a task is run cooperatively (rather than by interrupts), it is
    run when the
 *    method schedule() is called from within the main while loop in the
    main()
 *    routine. The schedule() method belongs to class stl_task and
    generally does not
 *    need to be modified by the user.
 */

class stl_task
{
    private:
        /// This variable, shared by all tasks, counts serial numbers
            during creation
        static char serial_counter;

        /// This is the automatically assigned serial number of this task
        char serial_number;

        /// This is the operational state (running, suspended, etc.) of
            this task
        task_op_state op_state;

        /// This saves the previous operational state of a suspended task
        task_op_state save_op_state;

        /// This is the state (as seen by the user) in which this task is
            right now
        char current_state;

    protected:
        /// This is a reference to the device driver which keeps track of
            real time
        task_timer& the_timer;

        /// This is the time at which the task should next be run
        time_stamp next_run_time;

        /// This is the time interval between runs of the task
        time_stamp interval;

    public:
        // The constructor sets time interval between runs and debug port
```

```
    (if used)
stl_task (task_timer&, const time_stamp&);

// This metod sets or changes the time interval between runs of the
    task
void set_interval (const time_stamp&);

// This method sets the next time the task is to run
void set_next_run_time (const time_stamp&);

// This method is called to give a task an opportunity to run its
    run() method
virtual task_op_state schedule (time_stamp* = NULL);

virtual char run (char);         // Base method which the user
    overloads
void suspend (void);             // Set operational state to
    suspended
void resume (void);              // Un-suspend a task so it can run
    again
void set_initial_state (char);   // Set a new state in which to
    start up

/** This method returns the task's automatically assigned serial
    number.
 *  @return The task's serial number
 */
char get_serial_number (void) { return (serial_number); }

/** This method returns the state in which the task is currently.
    The state
 *  cannot directly be changed by the user; it can only be changed
     through
 *  the returned value from the run() method.
 *  @return The state in which the task is when this method is
     called
 */
char get_current_state (void) { return (current_state); }

/** This method returns the task's current operational state. The
    operational
 *  state isn't the same as the state transition logic state; it's
     a separate
 *  variable which controls if the task is running at a given time.
 *  @return The task's current operational state
 */
task_op_state get_op_state (void) { return (op_state); }

/** This method will cause the task to run again as soon as it can
    instead of
 *  waiting for the given time interval.
```

```
     */
    inline void run_again_ASAP (void) { op_state = TASK_PENDING; }

    /** This method tells whether the task needs to run again as soon
        as possible
     *  or not. It is convenient to use when determining if the
         processor should
     *  be put to sleep for a while.
     *  @return True if the task needs to run soon, false if it does not
     */
    inline bool ready (void)
        { return (op_state == TASK_PENDING || op_state ==
            TASK_RUNNING); }

    void error_stop (char const*);  // Complain and stop the processor

// The following block is only compiled if execution time profiling
    has been
// enabled for this project by setting -DSTL_PROFILE in the Makefile
#ifdef STL_PROFILE
    protected:
        time_stamp max_time;         ///< Maximum time function took to
            run
        time_stamp min_time;         ///< Time for the quickest
            function run
        time_stamp avg_time;         ///< Stamp holds recently computed
            average
        time_stamp run_time_sum;     ///< Sum of measured function run
            times
        uint32_t runs;               ///< Number of runs that have been
            measured
        time_stamp start_time;       ///< Time at which measurement was
            started

    public:
        void clear_profiler (void);  // Clear statistical data items
        void start_profiler (void);  // Begin a measurement run
        void end_profiler (void);    // End a measurement and record
            times

        // This method returns the duration of the longest recorded run
            of the
        // code which is being profiled.
        const time_stamp& get_max_run_time (void);

        // This method returns the duration of the shortest recorded
            run of the
        // code which is being profiled.
        const time_stamp& get_min_run_time (void);

        // This method returns the average duration of all the runs of
```

```
            the code
        // which is being profiled.
        const time_stamp& get_avg_run_time (void);

        /** This method returns the number of runs of run() whose
            execution time
         *  has been measured in the current sample set.
        *  @return The number of runs which have been measured
        */
        uint32_t get_num_runs (void) { return (runs); }
    #endif // STL_PROFILE
};

// This overloaded operator prints information about a task to a serial
    device
base_text_serial& operator<< (base_text_serial&, stl_task&);

#endif // _STL_TASK_H_
```

Code Block G.19: State Transition Logic Task Class stl_task.cpp

```
//*****************************************************************************
/** \file stl_task.cpp
 *    This file contains the definition of a task class. This class
    implements the
 *    tasks in a multitasking system. These tasks are intended to be run in
    a
 *    cooperative multitasking framework, but they can also be worked into
    an
 *    interrupt framework or run within an RTOS.
 *
 *  Revisions
 *    \li 04-21-2007 JRR Original copy of this file, derived from UCB's
    TranRun4 and
 *                       simplified greatly for efficient use in AVR
    processors
 *    \li 05-07-2007 JRR Small bug fixes
 *    \li 06-01-2008 JRR Changed debugging/trace to take advantage of
    base_text_serial
 *    \li 06-03-2008 JRR Cleaned up comments, got rid of Doxygen warnings
 *    \li 12-19-2009 JRR Integrated simple execution time profiling into
    file, changed
 *                       from *.cc to *.cpp, and set up for global serial
    debugging
 *
 *  License:
 *    This file released under the Lesser GNU Public License, version 2.
    This program
 *    is intended for educational use only, but it is not limited thereto.
 */
//*****************************************************************************

#include <stdlib.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include "base_text_serial.h"          // Base class for various serial
   devices
#include "global_debug.h"              // Class for serial port debugging
#include "stl_timer.h"                 // Timer measures real time
#include "stl_task.h"                  // The state transition logic
   header


//---------------------------------------------------------------------------
// This is the static member declaration for class stl_task; see class
   definition in
// the header file for more information on the item(s)

char stl_task::serial_counter = 0;
```

```
/// This is just a time stamp with zeros in it for comparisions
const time_stamp zero_time (0L);


//---------------------------------------------------------------------------
/** This constructor creates a task object. It must be called by the
    constructor of
 *  each task which is written by the user. This constructor sets the time
     between runs
 *  of the run() method (a timestamp with zero time can be used if a task
     needs to run
 *  via interrupt, or every single time its schedule() method is called).
     This
 *  constructor also saves a pointer to the serial port or radio object
     which will be
 *  used to dump debugging information, if serial debugging is being used.
 *  @param a_timer A reference to the timer which measures real-time
     intervals
 *  @param time_interval The time between runs of the task's run() method
 */

stl_task::stl_task (task_timer& a_timer, const time_stamp& time_interval)
    : the_timer (a_timer), interval (time_interval)
{
    // Give this task its serial number, then increment the serial number
        counter
    serial_number = serial_counter++;

    // This task begins life in its waiting to run operational state
    op_state = TASK_WAITING;
    save_op_state = TASK_WAITING;

    // The task begins running in state 0, with no transitions unless
        called for
    current_state = 0;

    // The next run time should have been initialized to zero, so the task
        will run
    // its run() method as soon as possible in most cases

    #ifdef STL_PROFILE
        // Clear the profile data arrays
        clear_profiler ();
    #endif
}


//---------------------------------------------------------------------------
/** This method sets or changes the time interval between runs of this
    task.
 *  @param time_interval The time between runs of the task's run() method
```

```
 */

void stl_task::set_interval (const time_stamp& time_interval)
{
    interval = time_interval;
}


//---------------------------------------------------------------------------
/** This method sets the next run time for the task. It should be used if
    the task
 *  timer's value is adjusted significantly, or if the task timer is
     started at a time
 *  long after time zero. If this isn't done, the task will run many times
     in quick
 *  succession, trying to catch up to the correct run time. Note that if
     this method
 *  is called and sets the next time far in the past, the same problem
     will occur.
 *  @param next_time A time stamp containing the next time for the task to
     run
 */

void stl_task::set_next_run_time (const time_stamp& next_time)
{
    next_run_time = next_time;
}


//---------------------------------------------------------------------------
/** This method is called by the main task loop to try to run the task. If
    the task is
 *  in the waiting state, it checks to see if it's time to run yet; if
     it's in the
 *  suspended state, it doesn't. The task shouldn't be in the running
     state, because
 *  this method is used by the cooperative scheduler, not the pre-emptive
     one.
 *  @param til_next_time A pointer to a time stamp which will hold next
     time this task
 *                       next needs to run, or NULL (default) if this data
     isn't needed
 *  @return The operational state of the task after it has been given this
     chance to
 *         run
 */

task_op_state stl_task::schedule (time_stamp* til_next_time)
{
    char next_state;                    // State to which a task will
       transition
```

```
time_stamp the_time;                // Somewhere to store a time
    measurement

switch (op_state)
{
    // If the task has been suspended, don't bother trying to run it,
        and don't
    // bother sending back the next run time because this task is
        suspended
    case (TASK_SUSPENDED):
        return (TASK_SUSPENDED);

    // If the task needs to run, check if it needs to run now; if so,
        run it
    case (TASK_WAITING):
        // Find out what the current real time is from the system timer
        the_time = the_timer.get_time_now ();

        // If it's not time to run the task yet, exit without running it
        if (next_run_time > the_timer.get_time_now ())
        {
            if (til_next_time != NULL)
            {
                *til_next_time = next_run_time;
            }
            return (TASK_WAITING);
        }
        // If we get here, it is time to run the task; just continue
            into the
        // task_pending section below, which will cause the task to run
            right now

    case (TASK_PENDING):
        // Set the state to waiting for the next time interval. If the
            task needs
        // to run again immediately, run_again_ASAP() will be called
            within the
        // run() method, causing the state to be set to TASK_PENDING
            instead
        op_state = TASK_WAITING;
        #ifdef STL_PROFILE                  // If execution time
            profiling is
            start_profiler ();              // activated, start timing
        #endif
        next_state = run (current_state);   // Call the run() method
        #ifdef STL_PROFILE
            end_profiler ();                // End execution time
                measurement
        #endif
        if (next_state != STL_NO_TRANSITION) // Detect state transition
            if any
```

```
            {                                       // has occurred
                #ifdef STL_TRACE
                    GLOB_DEBUG ("T" << serial_number << ":" << current_state
                        << "-"
                        << next_state << endl);
                #endif
                current_state = next_state;     // Go to next state next
                    time
            }

            // If the op-state is TASK_WAITING now, then run_again_ASAP()
                was not
            // called, and the task will wait until its time to run again
                has arrived
            if (op_state == TASK_WAITING)
            {
                next_run_time += interval;
                if (til_next_time != NULL)
                {
                    *til_next_time = next_run_time;
                }
            }

            // If the op-state isn't TASK_WAITING, someone called
                run_again_ASAP(), so
            // the op-state is TASK_PENDING and we need to run again right
                away
            return (op_state);

        // If the operational state is anything else, there has been a
            serious error
        default:
            error_stop ("Illegal operational state");
            break;
    };

//   else if (op_state == blocked)              // Blocked tasks are not
    currently
//      return (false);                         // implemented

}


//-------------------------------------------------------------------------------
/** This is a base method which the user should overload in each
    descendent of this
 *  task class. The run method is where all the user-defined action in the
     task takes
 *  place. It is either called by the task's schedule() method, in the
     case of
 *  cooperative multitasking, or by an interrupt handler, in the case of
```

```
    pre-emptive
 *  multitasking.
 *  @param a_state The state of the task when this run method begins
    running
 *  @return The state to which the task will transition, or
    STL_NO_TRANSITION if no
 *   transition is called for at this time
 */

char stl_task::run (char a_state)
{
    GLOB_DEBUG (PMS ("Base run() method called for task ") <<
       serial_number << endl);

    return (STL_NO_TRANSITION);
}


//-------------------------------------------------------------------------------
/** This method suspends the task so that it won't run again until after
    somebody calls
 *  the resume() method. The state of the task before suspension is saved
    so that it
 *  can be restored after suspension.
 */

void stl_task::suspend (void)
{
    save_op_state = op_state;
    op_state = TASK_SUSPENDED;
}


//-------------------------------------------------------------------------------
/** This method resumes the task from suspension, so that the task can run
    again. The
 *  operational state which was saved at the time of suspension is
    restored.
 */

void stl_task::resume (void)
{
    op_state = save_op_state;
}


//-------------------------------------------------------------------------------
/** This method changes the initial state in which the task begins to
    operate. The
 *  default initial state is state 0. It should only be used before the
    task begins to
```

```
 *  run, as it sets the current and next state variables and will
    completely mess up
 *  normal state transition operation if called at any other time.
 */

void stl_task::set_initial_state (char init_state)
{
    current_state = init_state;
}


//-----------------------------------------------------------------------------
/** This method displays a message (if the program was compiled with
    serial debugging
 *  enabled) and then causes the processor to freeze in an infinite loop.
    It should be
 *  used if something awful happened and the safest thing to do is to just
    stop. Only
 *  use this function if there isn't a reasonable way to write an error
    state which
 *  handles exceptions in a more useful manner, such as by turning motors
    and other
 *  possibly dangerous devices off and then halting.
 *  @param message The text to be displayed before the processor stops
    working
 */

void stl_task::error_stop (char const* message)
{
    GLOB_DEBUG (PMS ("ERROR in task ") << serial_number << PMS (" state ")
        << current_state << ": " << message << endl << PMS ("Processing
            stopped.")
        << endl);

    cli ();                             // Disable interrupts
    while (1);                          // Bang...you're dead (until reset)
}


#ifdef STL_PROFILE
//-----------------------------------------------------------------------------
/** This method clears the statistical counters so that the profiler is
    ready to save
 *  some new data.
 */

void stl_task::clear_profiler (void)
{
    time_stamp zero (0, 0);
    max_time = zero;
    min_time = zero;
```

```
    avg_time = zero;
    run_time_sum = zero;
    runs = 0L;
}


//--------------------------------------------------------------------------------
/** This method begins a measurement run. It simply saves the time at
    which this
 *  method is called. A subsequent call to end() will cause the duration
     between the
 *  calls to start() and end() to be measured and recorded.
 */

void stl_task::start_profiler (void)
{
    start_time = the_timer.get_time_now ();
}


//--------------------------------------------------------------------------------
/** This method ends a measurement run, computing the duration of the run
    and saving
 *  that duration in the statistical counters as necessary.
 */

void stl_task::end_profiler (void)
{
    // Compute the duration measured for this run
    time_stamp duration = the_timer.get_time_now () - start_time;

    // A duration less than zero is impossible, so it must be wrong
    if (duration < 0)
    {
        return;
    }

    // If we have a maximum record, save it
    if (duration > max_time)
        max_time = duration;

    // If this is the first run, it's the minimum; otherwise, check it
    if (runs == 0L)
        min_time = duration;
    else
        if (duration < min_time)
            min_time = duration;

    // Add to the counters which are used to find the average
    run_time_sum += duration;
    runs++;
```

```
}


//---------------------------------------------------------------------------
/** This method returns the duration of the longest recorded run of the
    code
 *  which is being profiled.
 *  @return The maximum measured run duration
 */

const time_stamp& stl_task::get_max_run_time (void)
{
    return (max_time);
}


//---------------------------------------------------------------------------
/** This method returns the duration of the shortest recorded run of the
    code
 *  which is being profiled.
 *  @return The minimum measured run duration
 */

const time_stamp& stl_task::get_min_run_time (void)
{
    return (min_time);
}


//---------------------------------------------------------------------------
/** This method returns the average duration of all the runs of the code
    which
 *  is being profiled.
 *  @return The average measured run duration
 */

const time_stamp& stl_task::get_avg_run_time (void)
{
    if (runs > 0L)
    {
        avg_time = run_time_sum;
        avg_time /= runs;
    }
    return (avg_time);
}

#endif // STL_PROFILE


//---------------------------------------------------------------------------
/** This overloaded shift operator writes the vital statistics for this
```

```
   task to a
 *  serial device. Just what is written depends on whether transition
    tracing and/or
 *  execution time profiling has been enabled.
 *  @param serial A reference to the serial-type object to which to print
 *  @param task A reference to the task whose information is to be
    displayed
 */

base_text_serial& operator<< (base_text_serial& serial, stl_task& task)
{
    serial << PMS ("Task: ") << task.get_serial_number ();

    #ifdef STL_PROFILE
        serial << PMS (" runs: ") << task.get_num_runs ();
        time_stamp fred = task.get_avg_run_time ();
        serial << PMS (" avg: ") << fred;
        fred = task.get_max_run_time ();
        serial << PMS (" max: ") << fred;
        fred = task.get_min_run_time ();
        serial << PMS (" min: ") << fred;
    #endif

    return (serial);
}
```

Code Block G.20: Character Queue Header queue.h

```
//*****************************************************************************
/** \file queue.h
 *   This file implements a simple queue (or circular buffer). Data which
    is placed in
 *   the queue is saved for future reading. The queue class is implemented
    as a
 *   template; this means that a queue can be configured to store data of
    almost any
 *   type.
 *
 *  Usage:
 *   The template specifies the type of data being stored as qType, the
    type of
 *   integer used for the index as qIndexType, and the number of data
    items in the
 *   queue as qSize. First choose the type of data being stored; this may
    be simple
 *   data types such as characters or integers, but it may also be
    something more
 *   complex such as objects of a class (if there's enough memory to hold
    them all).
 *   Then look at the number of items in the queue. If there are less than
    255, use
 *   a character as qIndexType; otherwise use a regular 16-bit integer.
    For example,
 *   a queue holding 100 long integers (which would use 400 bytes for the
    data plus
 *   a few more bytes for the indices) could be declared as
 *   "queue<long, char, 100> my_queue".
 *
 *  Revisions:
 *   \li 08-28-03 JRR Reference: www.yureu.com/logo_plotter/queue_code.htm
 *   \li 08-28-03 JRR Ported to C++ from C
 *   \li 12-14-07 JRR Doxygen comments added
 *   \li 02-17-08 JRR Changed index type from define to template parameter
 *   \li 08-12-08 JRR Added overloaded array subscript operator
 *
 *  License:
 *   This file copyright 2007-2008 by JR Ridgely. It is released under the
    Lesser GNU
 *   public license, version 2. It is intended for educational use only,
    but the
 *   user is free to use it for any purpose permissible under the LGPL.
    The author
 *   has no control over the use of this file and cannot take any
    responsibility
 *   for any consequences of this use.
 */
//*****************************************************************************
```

```
/// These defines prevent this file from being included more than once in
    a *.cc file
#ifndef _QUEUE_H_
#define _QUEUE_H_
    // s

//-----------------------------------------------------------------------------
/** This class implements a simple queue. As a template class, it can be
    used to
 *  create queues which hold any type of object whose data is copied in
     with an
 *  assignment operator. The size and type of object are determined at
     compile time
 *  for efficiency.
 */

template <class qType, class qIndexType, qIndexType qSize>
class queue
{
    protected:
        qType buffer[qSize];         ///< This memory buffer holds the
            contents
        qIndexType i_put;           ///< Index where newest data will be
            written
        qIndexType i_get;           ///< Index where oldest data was written
        qIndexType how_full;         ///< How many elements are full at
            this time

    public:

        queue (void);             // Constructor
        bool put (qType);         // Adds one item into queue
        bool jam (qType);         // Force entry even if queue full
        qType get (void);         // Gets an item from the queue
        bool is_empty (void);     // Is the queue empty or not?
        void flush (void);        // Empty out the whole buffer
        void delete_one (void);   // Delete the last entry

        // This operator returns an item at the given index in the queue
        qType operator[] (qIndexType);

        /** This method returns the number of items in the queue */
        qIndexType num_items (void) { return (how_full); }

        /** This method returns a pointer to the buffer which holds the
            data in the
         *  queue. Normally it's not used, but in some cases (particularly
             queues of
         *  characters) one may need to access the data buffer directly.
         *  @return A pointer to the buffer holding the data in the queue
         */
```

```
        qType* get_p_buffer (void) { return buffer; }
};


//---------------------------------------------------------------------------
/** This constructor creates a queue item. Note that it doesn't have to
    allocate
 *  memory because that was already done statically by the template
     mechanism at
 *  compile time. If no parameters are supplied, a 16-element queue for
     characters
 *  which uses 8-bit array indices (for a maximum of 255 elements) is
     created.
 *  The template parameters are as follows:
 *  \li qType: The type of data which will be stored in the queue (default
     char)
 *  \li qIndexType: The type of data used for array indices
 *  \li qSize: The number of items in the queue's memory buffer (default
     16)
 */

template <class qType, class qIndexType, qIndexType qSize>
queue<qType, qIndexType, qSize>::queue (void)
{
    flush ();
}


//---------------------------------------------------------------------------
/** This method empties the buffer. It doesn't actually erase everything;
    it just sets
 *  the indices and fill indicator all to zero as if the buffer contained
     nothing.
 */

template <class qType, class qIndexType, qIndexType qSize>
void queue<qType, qIndexType, qSize>::flush (void)
{
    i_put = 0;
    i_get = 0;
    how_full = 0;
}


//---------------------------------------------------------------------------
/** This method adds an item into the queue. If the buffer is full then
    nothing is
 *  written and true (meaning buffer is full) is returned. This allows the
     calling
 *  program to see that the data couldn't be written and try again if
     appropriate.
```

```
 *  @param data The data to be written into the queue
 *  @return True if the buffer was full and data was not written, false
     otherwise
 */

template <class qType, class qIndexType, qIndexType qSize>
bool queue<qType, qIndexType, qSize>::put (qType data)
{
    // Check if the buffer is already full
    if (how_full >= qSize)
        return true;

    // OK, there's room in the buffer so add the data in
    buffer[i_put] = data;
    if (++i_put >= qSize)
        i_put = 0;
    how_full++;

    return false;
}


//---------------------------------------------------------------------------
/** This method jams an item into the queue whether the queue is empty or
    not. This
 *  can overwrite existing data, so it must be used with caution.
 *  @param data The data to be jammed into the queue
 *  @return True if some data was lost, false otherwise
 */

template <class qType, class qIndexType, qIndexType qSize>
bool queue<qType, qIndexType, qSize>::jam (qType data)
{
    // Write the data and move the write pointer to the next element
    buffer[i_put] = data;
    if (i_put >= qSize)
        i_put = 0;

    // Check if the buffer is already full; if so, the read index has to
        be moved so
    // that it points to the oldest unread data, which isn't the data
        written now
    if (how_full >= qSize)
    {
        if (++i_get >= qSize)
            i_get = 0;
        return true;
    }
    else
    {
        how_full++;                   // If we get here, the buffer isn't
```

```
            full
        return false;
    }
}


//-----------------------------------------------------------------------------
/** This method returns the oldest item in the queue. If the queue was
    empty, this is
 *  already-retreived data. Somebody should have checked if there was new
     data
 *  available using the is_empty() method.
 *  @return The data which is pulled out from the queue at the current
     location
 */

template <class qType, class qIndexType, qIndexType qSize>
qType queue<qType, qIndexType, qSize>::get (void)
{
    qType whatIgot;                    // Temporary storage for what was read

    whatIgot = buffer[i_get];            // Read and hold the data

    if (how_full > 0)                    // If the buffer's not empty,
    {                                  // moveflush the read pointer to the
        next full
        if (++i_get >= qSize)            // element
            i_get = 0;
        how_full--;                    // There's now one less item in the
            buffer
    }

    return (whatIgot);
}


//-----------------------------------------------------------------------------
/** This method returns true if the queue is empty. Empty means that there
    isn't
 *  any data which hasn't previously been read.
 *  @return True if the queue has unread data, false if it doesn't
 */

template <class qType, class qIndexType, qIndexType qSize>
bool queue<qType, qIndexType, qSize>::is_empty (void)
{
    return (how_full == 0);
}


//-----------------------------------------------------------------------------
```

```
/** This overloaded array subscript operator implements subscripts in the
    queue. The
 *  index is defined relative to the location at which the oldest data in
     the queue
 *  was written. This operator returns the (index)-th item in the queue,
     or (-1)
 *  typecast to the queue data type if there aren't enough items in the
     queue to get
 *  to the given index. The retreived data is not removed from the queue.
 *  @param index The array index into the queue at which to get a data item
 *  @return The data which is at the given index
 */

template <class qType, class qIndexType, qIndexType qSize>
qType queue<qType, qIndexType, qSize>::operator[] (qIndexType index)
{
    // Check if there's data written at the given location
    if (index > how_full)
        return ((qType)(-1));

    // Find an index pointing to the correct location in the queue
    qIndexType getIndex = i_get + index;
    if (getIndex > qSize)
        getIndex -= qSize;

    // Get the data at that index location
    return (buffer[getIndex]);
}

//-----------------------------------------------------------------------------
/** This method decrements the i_put and how_full variables to "delete"
    the latest character.
 */

template <class qType, class qIndexType, qIndexType qSize>
void queue<qType, qIndexType, qSize>::delete_one (void)
{
    i_put--;
    how_full--;
}

#endif // _QUEUE_H_
```

Code Block G.21: Character Queue Class queue.cpp

```
//*****************************************************************************
//  File: queue.cpp
//      This file contains a class which implements a simple queue. This
    compiles for
//      the AVR platform with avr-gcc.
//
//  Version:
//      Original file from http://www.yureu.com/logo_plotter/queue_code.htm
//       08-28-03 JRR  Ported to C++ from C
//       04-17-08 JRR  Changed from avr_queue to just queue
//
//*****************************************************************************

#include "queue.h"

//  Everything's in the header file, actually. This file is just here to
//  allow an entry into the makefile, if that's even needed
```

Code Block G.22: Base Generic Serial Header base232.h

```
//*****************************************************************************
/** \file base232.h
 *   This file contains a set of macros which are used by several classes
    and class
 *   templates that interact with the asynchronous (RS-232 style) serial
    port on an
 *   AVR microcontroller. Classes which use this stuff include rs232 and
    packet232.
 *
 *  Revisions:
 *    \li 04-03-2006 JRR For updated version of compiler
 *    \li 06-10-2006 JRR Ported from C++ to C for use with some C-only
    projects; also
 *        serial_avr.h now has defines for compatibility among lots of AVR
    variants
 *    \li 08-11-2006 JRR Some bug fixes
 *    \li 03-02-2007 JRR Ported back to C++. I've had it with the
    limitations of C.
 *    \li 04-16-2007 JO Added write (unsigned long)
 *    \li 07-19-2007 JRR Changed some character return values to bool,
    added m324p
 *    \li 01-12-2008 JRR Added code for the ATmega128 using USART number 1
    only
 *    \li 02-14-2008 JRR Split between base_text_serial and rs232 files
 *    \li 05-31-2008 JRR Changed baud calculations to use CPU_FREQ_MHz from
    Makefile
 *    \li 06-01-2008 JRR Added getch_tout() because it's needed by 9Xstream
    modems
 *    \li 07-05-2008 JRR Changed from 1 to 2 stop bits to placate finicky
    receivers
 *    \li 12-22-2008 JRR Split off stuff in base232.h for efficiency
 *    \li 01-30-2009 JRR Added class with port setup in constructor
 *    \li 06-02-2009 JRR Changed baud rate divisor formula to work better
 *    \li 12-14-2009 JRR Changed CPU_FREQ_Hz to F_CPU to be compatible with
    avr-libc
 *
 *  License:
 *   This file is released under the Lesser GNU Public License, version 2.
    This
 *   program is intended for educational use only, but it is not limited
    thereto.
 */
//*****************************************************************************

/// This define prevents this .h file from being included more than once
   in a .cc file
#ifndef _BASE232_H_
#define _BASE232_H_

#define F_CPU  20000000
```

```
// #include "base_text_serial.h"           // Pull in the base class
    header file

// Check that the user has set the CPU frequency in the Makefile; if not,
    complain
#if defined (__AVR) && !defined (F_CPU)
    #error The macro F_CPU must be set in the Makefile
#endif

//-----------------------------------------------------------------------------
/** The number of tries to wait for the transmitter buffer to become empty
    */
#define UART_TX_TOUT     20000

//-----------------------------------------------------------------------------
/** If this macro is defined, the UART will run in double-speed mode. This
    is often
 *  a good idea, as it allows higher baud rates to be used with not so
     high CPU clock
 *  frequencies.
 */
#define UART_DOUBLE_SPEED

//-----------------------------------------------------------------------------
/** This macro computes a value for the baud rate divisor from the desired
    baud rate
 *  and the CPU clock frequency. The CPU clock frequency should have been
     set in the
 *  macro F_CPU, which is normally configured in the Makefile. The divisor
     is
 *  calculated as (frequency, MHz / (16 * baudrate)), unless the USART is
     running in
 *  double-speed mode, in which case the baud rate divisor is twice as big.
 */

#ifdef UART_DOUBLE_SPEED
    #define calc_baud_div(baud_rate) ((F_CPU) / (16UL * (baud_rate)))
#else
    #define calc_baud_div(baud_rate) ((F_CPU) / (8UL * (baud_rate)))
#endif


//-----------------------------------------------------------------------------
/** This class operates the hardware on an RS232 port in an AVR
    microcontroller. It
 *  sets up the port, checks for characters, and sends characters one at a
     time. Its
 *  descendent classes do useful things such as sending text messages and
     packets.
 */
```

```
class base232
{
    protected:
    #ifdef __AVR
        /// This is a pointer to the data register used by the UART
        volatile unsigned char* p_UDR;

        /// This is a pointer to the status register used by the UART
        volatile unsigned char* p_USR;

        /// This is a pointer to the control register used by the UART
        volatile unsigned char* p_UCR;

        /// This bitmask identifies the bit for data register empty, UDRE
        unsigned char mask_UDRE;

        /// This bitmask identifies the bit for receive complete, RXC
        unsigned char mask_RXC;

        /// This bitmask identifies the bit for transmission complete, TXC
        unsigned char mask_TXC;
    #else
        /// This is the file handle for the serial port file device on a PC
        int serial_file;
    #endif // __AVR

    public:
    #ifdef __AVR
        /// The constructor sets up the port with the given baud rate and
           port number.
        base232 (unsigned int = 9600, unsigned char = 0);
    #else
        /// The constructor sets up the port with the given name.
        base232 (char*);
    #endif

        /// This method checks if the serial port is ready to transmit data.
        bool ready_to_send (void);

        /// This method returns true if the port is currently sending a
           character out.
        bool is_sending (void);
};

#endif // _BASE232_H_
```

Code Block G.23: Base Generic Serial Class base232.cpp

```
//*****************************************************************************
/** \file base232.cpp
 *   This file contains a class which operates an asynchronous (RS-232
    style) serial
 *   port on an AVR microcontroller. Classes which use this stuff include
    rs232 and
 *   packet232.
 *
 *  Revisions:
 *    \li 04-03-2006 JRR For updated version of compiler
 *    \li 06-10-2006 JRR Ported from C++ to C for use with some C-only
    projects; also
 *        serial_avr.h now has defines for compatibility among lots of AVR
    variants
 *    \li 08-11-2006 JRR Some bug fixes
 *    \li 03-02-2007 JRR Ported back to C++. I've had it with the
    limitations of C.
 *    \li 04-16-2007 JO Added write (unsigned long)
 *    \li 07-19-2007 JRR Changed some character return values to bool,
    added m324p
 *    \li 01-12-2008 JRR Added code for the ATmega128 using USART number 1
    only
 *    \li 02-14-2008 JRR Split between base_text_serial and rs232 files
 *    \li 05-31-2008 JRR Changed baud calculations to use CPU_FREQ_MHz from
    Makefile
 *    \li 06-01-2008 JRR Added getch_tout() because it's needed by 9Xstream
    modems
 *    \li 07-05-2008 JRR Changed from 1 to 2 stop bits to placate finicky
    receivers
 *    \li 12-22-2008 JRR Split off stuff in base232.h for efficiency
 *    \li 01-30-2009 JRR Added class with port setup in constructor
 *    \li 06-02-2009 JRR Changed baud rate divisor formula to work better
 *
 *  License:
 *   This file is released under the Lesser GNU Public License, version 2.
    This
 *   program is intended for educational use only, but it is not limited
    thereto.
 */
//*****************************************************************************

#ifdef __AVR
   #include <avr/io.h>                 // Definitions of AVR's I/O
      registers
   #include "global_debug.h"           // For a global debugging port
#else
   #include <stdlib.h>                 // Standard stuff such as exit()
   #include <fcntl.h>                  // File devices on Linux computers
   #include <iostream>                 // Headers for I/O streams so we
      can use
```

```
    #include <iomanip>                  // 'cout << stuff' style printing
    using namespace std;                // Use file I/O from standard
        library
#endif

#include "base232.h"


//-------------------------------------------------------------------------------
/** This method sets up the AVR UART for communications. It enables the
    appropriate
 *  inputs and outputs and sets the baud rate divisor, and it saves
     pointers to the
 *  registers which are used to operate the serial port. Since some AVR
     processors
 *  have dual serial ports, this method allows one to specify a port
     number.
 *  @param baud_rate The desired baud rate for serial communications.
     Default is 9600
 *  @param port_number The number of the serial port, 0 or 1 (the second
     port numbered
 *                   1 only exists on some processors). The default is
     port 0
 */

// This section compiles for the AVR microcontroller
#ifdef __AVR
base232::base232 (unsigned int baud_rate, unsigned char port_number)
{
    // If we're compiling for a chip with UCSR0A defined, it has dual
        serial ports
    // (examples are ATmega324P and ATmega128). Set up Port 0 or Port 1
    #if defined UCSR0A
        if (port_number == 0)
        {
            p_UDR = &UDR0;
            p_USR = &UCSR0A;
            p_UCR = &UCSR0B;
            UCSR0B = (1 << RXEN0) | (1 << TXEN0);
            UCSR0C = (1 << UCSZ01) | (1 << UCSZ00); // | (1 << USBS0);
            UBRR0H = 0x00;
            UBRR0L = calc_baud_div (baud_rate);
            #ifdef UART_DOUBLE_SPEED                // Activate double speed
                mode
                UCSR0A |= U2X0;                     // if required
            #endif
            mask_UDRE = (1 << UDRE0);
            mask_RXC = (1 << RXC0);
            mask_TXC = (1 << TXC0);
        }
        else   // The port number isn't 0, so it presumably must be serial
```

```
        port 1
    {
    #if defined UCSR1A
        p_UDR = &UDR1;
        p_USR = &UCSR1A;
        p_UCR = &UCSR1B;
        UCSR1B = (1 << RXEN1) | (1 << TXEN1);
        UCSR1C = (1 << UCSZ11) | (1 << UCSZ10); // | (1 << USBS1);
        UBRR1H = 0x00;
        UBRR1L = calc_baud_div (baud_rate);
        #ifdef UART_DOUBLE_SPEED     // If double-speed macro has been
            defined,
            UCSR1A |= U2X1;          // turn on double-speed operation
        #endif
        mask_UDRE = (1 << UDRE1);
        mask_RXC = (1 << RXC1);
        mask_TXC = (1 << TXC1);
    #endif // UCSR1A
    }
// We're compiling for a chip which has no USCR0A, so it should have
    only one
// serial port. Ignore the port number and set up the only serial port
    available
#else // no UCSR0A
    // The more sophisticated chips, such as ATmega8 and ATmega32, have
        UCSRA
    // and UCSRB and UCSRC registers which are set up appropriately
    #ifdef UCSRA
        p_UDR = &UDR;
        p_USR = &UCSRA;
        p_UCR = &UCSRB;
        UCSRB = (1 << RXEN) | (1 << TXEN);
        UCSRC = (1 << URSEL) | (1 << UCSZ1) | (1 << UCSZ0);  // | (1 <<
            USBS0);
        UBRRH = 0x00;
        UBRRL = calc_baud_div (baud_rate);
        #ifdef UART_DOUBLE_SPEED            // Activate double speed
            mode
            UCSRA |= U2X;                   // if required
        #endif
        mask_UDRE = (1 << UDRE);
        mask_RXC = (1 << RXC);
        mask_TXC = (1 << TXC);
    // Older, simpler chips such as AT90S2313 have only the UCR register
    #else // no UCSRA
        p_UDR = &UDR;
        p_USR = &USR;
        p_UCR = &UCR;
        UCR = (1 << RXEN) | (1 << TXEN);    // 0x18 for mode N81
        UBRR = calc_baud_div (baud_rate);
        mask_UDRE = (1 << UDRE);
```

```
            mask_RXC = (1 << RXC);
            mask_TXC = (1 << TXC);
        #endif // UCSRA
    #endif // UCSR0A

    // Read the data register to ensure that it's empty
    port_number = *p_UDR;
    port_number = *p_UDR;
}
#else // If not AVR, we must be compiling for a Linux PC
base232::base232 (char* port_name)
{
    serial_file = open (port_name, O_RDWR | O_NOCTTY | O_NDELAY);
    if (serial_file <= 0)
    {
        cerr << "ERROR: Can't open serial port \"" << port_name << "\"" <<
            endl;
        exit (-1);
    }
    else
    {
        cout << "Opened serial port " << port_name << " as device " <<
            serial_file
             << endl;
    }
}
#endif // __AVR


//--------------------------------------------------------------------------------
/** This method checks if the serial port transmitter is ready to send
    data. It
 *  tests whether transmitter buffer is empty.
 *  @return True if the serial port is ready to send, and false if not
 */

bool base232::ready_to_send (void)
{
#ifdef __AVR
    // If transmitter buffer is full, we're not ready to send
    if (*p_USR & mask_UDRE)
        return (true);

    return (false);
#else
    // The non-AVR (that is, PC) serial port buffer should always be ready
    return (true);
#endif
}
```

```
//-----------------------------------------------------------------------------
/** This method checks if the serial port is currently sending a
    character. Even if the
 *  buffer is ready to accept a new character, the port might still be
     busy sending the
 *  last one; it would be a bad idea to put the processor to sleep before
     the character
 *  has been sent.
 *  @return True if the port is currently sending a character, false if
     it's idle
 */

bool base232::is_sending (void)
{
#ifdef __AVR
    if (*p_USR & mask_TXC)
        return (false);
    else
        return (true);
#else
    // We don't really care if a PC is sending, as it has a buffer anyway
    return (false);
#endif
}
```

Code Block G.24: Base Text Serial Header base_text_serial.h

```
//****************************************************************************
/** \file base_text_serial.h
 *   This file contains a base class for devices which send information in
    text form
 *   over serial devices in an AVR microcontroller. Example devices are
    serial ports
 *   (both traditional RS-232 ports and USB-serial adapters) and wireless
    modems.
 *
 *  Revised:
 *   \li 04-03-2006 JRR For updated version of compiler
 *   \li 06-10-2006 JRR Ported from C++ to C for use with some C-only
    projects
 *   \li 08-11-2006 JRR Some bug fixes
 *   \li 03-02-2007 JRR Ported back to C++. I've had it with the
    limitations of C.
 *   \li 04-16-2007 JO Added write (unsigned long)
 *   \li 07-19-2007 JRR Changed some character return values to bool,
    added m324p
 *   \li 01-12-2008 JRR Added code for the ATmega128 using USART number 1
    only
 *   \li 02-13-2008 JRR Split into base class and device specific classes;
    changed
 *                    from write() to overloaded << operator in the
    "cout" style
 *   \li 05-12-2008 ALS Fixed bug in which signed numbers came out unsigned
 *   \li 07-05-2008 JRR Added configuration macro by which to change what
    "endl" is
 *   \li 07-05-2008 JRR Added 'ascii' and 'numeric' format codes
 *   \li 11-24-2009 JRR Changed operation of 'clrscr' to a function to
    work with LCD
 *   \li 11-26-2009 JRR Integrated floating point support into this file
 *   \li 12-16-2009 JRR Improved support for constant strings in program
    memory
 *
 *  Licenses:
 *   This file released under the Lesser GNU Public License, version 2.
    This program
 *   is intended for educational use only, but it is not limited thereto.
    This code
 *   incorporates elements from Xmelkov's ftoa_engine.h, part of the
    avr-libc source,
 *   and users must accept and comply with the license of ftoa_engine.h as
    well.
 */
//----------------------------------------------------------------------------
/* The following is taken from ftoa_engine.h, part of the avr-libc source.
   It is subject to the following copyright notice:
   Copyright (c) 2005, Dmitry Xmelkov
   All rights reserved.
```

```
    Redistribution and use in source and binary forms, with or without
    modification, are permitted provided that the following conditions are
        met:

    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in
      the documentation and/or other materials provided with the
      distribution.
    * Neither the name of the copyright holders nor the names of
      contributors may be used to endorse or promote products derived
      from this software without specific prior written permission.

    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
        "AS IS" AND
    ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
        IMPLIED
    WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
        DISCLAIMED.
    IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR
        ANY DIRECT,
    INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
        (INCLUDING, BUT
    NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
        USE, DATA, OR
    PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
        LIABILITY,
    WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
        OR OTHERWISE)
    ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
        THE
    POSSIBILITY OF SUCH DAMAGE. */
//****************************************************************************

/// This define prevents this .h file from being included more than once
    in a .cc file
#ifndef _BASE_TEXT_SERIAL_H_
#define _BASE_TEXT_SERIAL_H_

#include <avr/pgmspace.h>               // Header for program-space
    (Flash) data

// Uncomment this line to enable floating point handling by
    base_text_serial; comment
// it out if you don't need floating point and would like to save lots of
    memory
#include <math.h>
```

```
// The following code block is only compiled if <math.h> has been included
#ifdef M_SQRT2
    extern "C"
    {
        int __ftoa_engine (double val, char *buf,
                           unsigned char prec, unsigned char maxdgs);
    }

    #define FTOA_MINUS    1             ///< Status code for a negative
        number
    #define FTOA_ZERO     2             ///< Status code for zero
    #define FTOA_INF      4             ///< Status code for an infinite
        result
    #define FTOA_NAN      8             ///< Status code for not-a-number
    #define FTOA_CARRY    16            ///< Status code for carrying
        something(?)
#endif // M_SQRT2


/** \brief This define selects what will be sent when a program sends out
    "endl".
 *  Different recieving programs want different end-of-line markers.
     Traditionally,
 *  UNIX uses "\r" while PC's use "\r\n" and Macs use "\n" (I think). */
#define ENDL_STYLE  "\r\n"

/** \brief This define selects the character which asks a terminal to
    clear its screen.
 *  This is usually a control-L, which is ASCII character number 12.
 */

#define CLRSCR_STYLE  ((unsigned char)12)

/** This define allows strings to be created in program memory and tells
    the puts()
 *  method to find them in program memory. This saves memory compared to
     copying all
 *  the strings to SRAM, which is the rather wasteful default treatment.
     See link(s),
 *  noting that you'll have to paste some multi-part links together
     because we run out
 *  of space in the comment box in the source code...
 *  \li
     http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=57011
        &start=all&postdays=0&postorder=asc
 */

#define PMS(s) _p_str << \
    __extension__({static const char __c[] PROGMEM = s; &__c[0]; })

//  __extension__({ static char __c[] PROGMEM = (s); &__c[0]; })
```

```
//-------------------------------------------------------------------------
/** This enumeration is used to change the display base for the output
    stream from the
 *  default of 10 (decimal) to and from 2 (binary), 8 (octal), or 16
     (hexadecimal).
 *  Also defined are endl for end-of-line and send_now to tell packet
     based devices to
 *  transmit a packet as soon as possible. The base is changed for the
     next printing
 *  of one number only, then the base reverts to the default value of base
     10. Codes
 *  are also provided to allow printing of 8-bit numbers as ASCII
     characters and to
 *  tell some devices to send packets of data immediately.
 */

typedef enum {
    bin,                  ///< Print following numbers in base 2 (binary)
    oct,                  ///< Print following numbers in base 8 (octal)
    dec,                  ///< Print following numbers in base 10 (decimal)
    hex,                  ///< Print following numbers in base 16
        (hexadecimal)
    ascii,                ///< Print an ASCII character rather than an
        8-bit number
    numeric,              ///< Print 8-bit numbers as numeric values, not
        ASCII
    endl,                 ///< Print a carriage return and linefeed
    clrscr,               ///< Send a control-L which clears some terminal
        screens
    send_now,             ///< Tell some devices to send or save data
        immediately
    manip_set_precision,  ///< Set the precision for printing floating
        point numbers
    _p_str                ///< The following string is in program (flash)
        memory
} ser_manipulator;


//-------------------------------------------------------------------------
// This function sets the number of digits to be printed after the decimal
    point, then
// notifies a serial port that it should set its precision accordingly

ser_manipulator setprecision (unsigned char);


//-------------------------------------------------------------------------
/** This is a base class for lots of serial devices which send text over
    some type of
```

```
 *  communication interface. Descendents of this class will be able to
    send text over
 *  serial lines, radio modems or modules, and whatever other
    communication devices
 *  might be appropriate.
 *
 *  The most important methods of this class are the overloaded "<<"
    operators which
 *  convert different types of numbers into printable strings, then print
    them using
 *  methods which are overloaded by descendent classes. Methods are
    provided which
 *  transmit most types of integers and floating point numbers; floating
    point numbers
 *  are only handled if \<math.h\> is defined near the beginning of
    base_text_serial.h
 *  because one often wants to turn off floating point support in order to
    save
 *  memory. Other classes may implement methods to print themselves to any
    descendent
 *  of this class; see stl_timer.h/cc for examples in which time stamps
    print the time
 *  conveniently to any serial device.
 *
 *  The ability to transmit characters is provided by descendent classes.
    Methods
 *  which will be inherited or overridden by descendents include the
    following:
 *    \li ready_to_send () - Checks if the port is ready to transmit a
    character
 *    \li putchar() - Sends a single character over the communications line
 *    \li puts() - Sends a character string
 *  Other methods may optionally be overridden; for example clear_screen()
    is only
 *  needed by devices which have a screen, and some devices may be
    read-only or
 *  write-only, and some methods may not be needed.
 */

class base_text_serial
{
   // Private data and methods are accessible only from within this class
       and
   // cannot be accessed from outside -- even from descendents of this
       class
   private:

   // Protected data and methods are accessible from this class and its
       descendents
   // only
   protected:
```

```
    /// This is the currently used base for converting numbers to text.
    unsigned char base;

    /** If this variable is set true, print characters as ASCII
        characters, not
     *  as numbers. */
    bool print_ascii;

    /** If this variable is true, the next string to be printed will be
        found in
     * program (flash) memory, not data memory. */
    bool pgm_string;

    /** This is the number of digits after a decimal point to be
        printed when a
     *  floating point number is being converted to text. */
    char precision;

// Public methods can be called from anywhere in the program where
    there is a
// pointer or reference to an object of this class
public:
    base_text_serial (void);        // Simple constructor doesn't do
        much
    virtual bool ready_to_send (void); // Virtual and not defined in
        base class
    virtual bool putchar (char) {}  ///< Virtual and not defined in
        base class
    virtual void puts (char const*) {} ///< Virtual and not defined in
        base class
    virtual bool check_for_char (void); // Check if a character is in
        the buffer
    virtual char getchar (void);    // Get a character; wait if none
        is ready
    virtual void transmit_now (void); // Immediately transmit any
        buffered data
    virtual void clear_screen (void); // Clear a display screen if
        there is one

    // The overloaded left-shift operators convert numbers to strings
        and send the
    // strings out the serial device; manipulators change the formatting
    base_text_serial& operator<< (bool);
    base_text_serial& operator<< (const char*);
    base_text_serial& operator<< (unsigned char);
    base_text_serial& operator<< (char num);
    base_text_serial& operator<< (unsigned int);
    base_text_serial& operator<< (int);
    base_text_serial& operator<< (unsigned long);
    base_text_serial& operator<< (long);
    base_text_serial& operator<< (unsigned long long);
```

```
        #ifdef M_SQRT2 // Automatically include this code if <math.h> has
            been included
            base_text_serial& operator<< (float);
            base_text_serial& operator<< (double);
        #endif
        base_text_serial& operator<< (ser_manipulator);
};

#endif // _BASE_TEXT_SERIAL_H_
```

Code Block G.25: Base Text Serial Class base_text_serial.cpp

```
//*****************************************************************************
/** \file base_text_serial.cpp
 *   This file contains a base class for devices which send information in
    text form
 *   over serial devices in an AVR microcontroller. Example devices are
    serial ports
 *   (both traditional RS-232 ports and USB-serial adapters) and wireless
    modems.
 *
 *  Revised:
 *   \li 04-03-2006 JRR For updated version of compiler
 *   \li 06-10-2006 JRR Ported from C++ to C for use with some C-only
    projects
 *   \li 08-11-2006 JRR Some bug fixes
 *   \li 03-02-2007 JRR Ported back to C++. I've had it with the
    limitations of C.
 *   \li 04-16-2007 JO Added write (unsigned long)
 *   \li 07-19-2007 JRR Changed some character return values to bool,
    added m324p
 *   \li 01-12-2008 JRR Added code for the ATmega128 using USART number 1
    only
 *   \li 02-13-2008 JRR Split into base class and device specific classes;
    changed
 *                    from write() to overloaded << operator in the
    "cout" style
 *   \li 05-12-2008 ALS Fixed bug in which signed numbers came out unsigned
 *   \li 07-05-2008 JRR Added configuration macro by which to change what
    "endl" is
 *   \li 07-05-2008 JRR Added 'ascii' and 'numeric' format codes
 *   \li 11-24-2009 JRR Changed operation of 'clrscr' to a function to
    work with LCD
 *   \li 11-26-2009 JRR Integrated floating point support into this file
 *   \li 12-16-2009 JRR Improved support for constant strings in program
    memory
 *
 *  Licenses:
 *   This file released under the Lesser GNU Public License, version 2.
    This program
 *   is intended for educational use only, but it is not limited thereto.
    This code
 *   incorporates elements from Xmelkov's ftoa_engine.h, part of the
    avr-libc source,
 *   and users must accept and comply with the license of ftoa_engine.h as
    well. See
 *   base_text_serial.h for a copy of the relevant license terms.
 */
//*****************************************************************************

#include <stdint.h>
#include <stdlib.h>
```

```
#include <avr/io.h>
#include "base_text_serial.h"


//-----------------------------------------------------------------------------
/** This variable, accessible to any function in this file, allows the
    nonmember
 *  function setprecision() to communicate the number of digits after the
     decimal
 *  point to a specific text serial port object. The setprecision()
     function will
 *  return a manipulator that causes a specific serial object (the one in
     whose cout
 *  style output line it is called) to read the value of this variable.
 */

unsigned char bts_glob_prec = 3;


//-----------------------------------------------------------------------------
/** This constructor sets up the base serial port object. It sets the
    default base for
 *  the conversion of numbers to text and the default format for
     converting chars.
 */

base_text_serial::base_text_serial (void)
{
    base = 10;                          // Numbers are shown as decimal by
        default
    print_ascii = false;                // Print 8-bit chars as numbers by
        default
    precision = 3;                      // Print 3 digits after a decimal
        point
    pgm_string = false;                 // Print strings from SRAM by
        default
}


//-----------------------------------------------------------------------------
/** This function checks if the serial port transmitter is ready to send
    data. It's
 *  an empty base method, overridden by most serial devices. Some serial
     devices might
 *  always be ready to send data; those can not bother to override this
     method.
 *  @return True if the serial port is ready to send, and false if not
 */

bool base_text_serial::ready_to_send (void)
{
```

```
    return (true);                      // By default the port's always
        ready
}


//-------------------------------------------------------------------------------
/** This base method just returns zero, because it shouldn't be called.
    There might be
 *  classes which only send characters and don't ever receive them, and
     this method
 *  could be left as a placeholder for those classes.
 *  @return A zero (null character) which means nothing
 */

char base_text_serial::getchar (void)
{
    return ('\0');                      // Nothing to return, really
}


//-------------------------------------------------------------------------------
/** This method checks if there is a character in the serial port's
    receiver buffer.
 *  There isn't, as this base class isn't connected to a buffer;
     descendent classes
 *  will override this method and check a real buffer for real characters.
 *  @return False, because no character will ever be available
 */

bool base_text_serial::check_for_char (void)
{
    return (false);
}


//-------------------------------------------------------------------------------
/** This is a base method for causing immediate transmission of a buffer
    full of data.
 *  The base method doesn't do anything, because it will be implemented in
     descendent
 *  classes which have no buffers, send everything immediately by default,
     and don't
 *  need to respond to calls for immediate transmission.
 */

void base_text_serial::transmit_now (void)
{
}


//-------------------------------------------------------------------------------
```

```
/** This is a base method to clear a display screen, if there is one. It
    is called
 *  when the format modifier 'clrscr' is inserted in an output line.
     Descendant
 *  classes which send things to displays should respond by clearing
     themselves.
 */

void base_text_serial::clear_screen (void)
{
}


//-----------------------------------------------------------------------------
/** This method writes the string whose first character is pointed to by
    the given
 *  character pointer to the serial device. It acts in about the same way
     as puts().
 *  As with puts(), the string must have a null character (ASCII zero) at
     the end.
 *  @param string Pointer to the string to be written
 */

base_text_serial& base_text_serial::operator<< (const char* string)
{
    // If the program-string variable is set, this string is to be found
       in program
    // memory rather than data memory
    if (pgm_string)
    {
        pgm_string = false;
        while (char ch = pgm_read_byte_near (string++))
            putchar (ch);

    }
    // If the program-string variable is not set, the string is in RAM and
       printed
    // in the normal way
    else
    {
        while (*string) putchar (*string++);
    }

    return (*this);
}


//-----------------------------------------------------------------------------
/** This method writes a boolean value to the serial port as a character,
    either "T"
 *  or "F" depending on the value.
```

```
 *  @param value The boolean value to be written
 */

base_text_serial& base_text_serial::operator<< (bool value)
{
    if (value)
        putchar ('T');
    else
        putchar ('F');

    return (*this);;
}


//-----------------------------------------------------------------------------
/** This method writes a character to the serial port as a text string
    showing the
 *  8-bit unsigned number in that character. If the format control term
     'ascii' was
 *  called prior to this printing, the character will be printed as an
     ASCII character.
 *  @param num The 8-bit number or character to be sent out
 */

base_text_serial& base_text_serial::operator<< (unsigned char num)
{
    unsigned char temp_char;            // Temporary storage for a nibble

    if (print_ascii)
    {
        putchar (num);
    }
    else if (base == 2)
    {
        for (unsigned char bmask = 0x80; bmask != 0; bmask >>= 1)
        {
            if (num & bmask) putchar ('1');
            else             putchar ('0');
        }
    }
    else if (base == 16)
    {
        temp_char = (num >> 4) & 0x0F;
        putchar ((temp_char > 9) ? temp_char + ('A' - 10) : temp_char +
            '0');
        temp_char = num & 0x0F;
        putchar ((temp_char > 9) ? temp_char + ('A' - 10) : temp_char +
            '0');
    }
    else
    {
```

```
        char out_str[9];
        utoa ((unsigned int)num, out_str, base);
        puts (out_str);
    }

    return (*this);
}


//-----------------------------------------------------------------------------
/** This method writes a character to the serial port as a text string
    showing the
 *  8-bit signed number in that character, unless the 'ascii' manipulator
     has been
 *  used to put the port in ascii text mode, in which case the character
     is printed
 *  directly.
 *  @param num The 8-bit number to be sent out
 */

base_text_serial& base_text_serial::operator<< (char num)
{
    char out_str[5];

    if (print_ascii)
        putchar (num);
    else
    {
        if (base == 10)
        {
            itoa ((int)num, out_str, 10);
            puts (out_str);
        }
        else
            *this << (unsigned char)num;
    }

    return (*this);
}


//-----------------------------------------------------------------------------
/** This method writes an integer to the serial port as a text string
    showing the
 *  16-bit unsigned number in that integer.
 *  @param num The 16-bit number to be sent out
 */

base_text_serial& base_text_serial::operator<< (unsigned int num)
{
    if (base == 16 || base == 8 || base == 2)
```

```
    {
        union {
            unsigned long whole;
            unsigned char bits[2];
        } parts;
        parts.whole = num;
        *this << parts.bits[1] << parts.bits[0];
    }
    else
    {
        char out_str[17];
        utoa (num, out_str, base);
        puts (out_str);
    }

    return (*this);
}


//-------------------------------------------------------------------------------------
/** This method writes an integer to the serial port as a text string
    showing the
 *  16-bit signed number in that integer.
 *  @param num The 16-bit number to be sent out
 */

base_text_serial& base_text_serial::operator<< (int num)
{
    if (base != 10)
    {
        *this << (unsigned int)num;
    }
    else
    {
        char out_str[17];

        itoa (num, out_str, base);
        puts (out_str);
    }

    return (*this);
}


//-------------------------------------------------------------------------------------
/** This method writes an unsigned long integer to the serial port as a
    text string
 *  showing the 32-bit unsigned number in that long integer.
 *  @param num The 32-bit number to be sent out
 */
```

```
base_text_serial& base_text_serial::operator<< (unsigned long num)
{
    if (base == 16 || base == 8 || base == 2)
    {
        union {
            unsigned long whole;
            unsigned char bits[4];
        } parts;
        parts.whole = num;
        *this << parts.bits[3] << parts.bits[2] << parts.bits[1] <<
            parts.bits[0];
    }
    else
    {
        char out_str[33];
        ultoa ((long)num, out_str, base);
        puts (out_str);
    }

    return (*this);
}


//-----------------------------------------------------------------------------
/** This method writes a long integer to the serial port as a text string
    showing the
 *  32-bit signed number in that long integer.
 *  @param num The 32-bit number to be sent out
 */

base_text_serial& base_text_serial::operator<< (long num)
{
    if (base != 10)
    {
        *this << (unsigned long)num;
    }
    else
    {
        char out_str[34];
        ltoa ((long)num, out_str, base);
        puts (out_str);
    }

    return (*this);
}


//-----------------------------------------------------------------------------
/** This method writes a long long (64-bit) unsigned integer to the serial
    port as a
 *  text string. It only writes such numbers in unsigned hexadecimal
```

```
    format. The
 *  number is written by breaking it into two unsigned longs, writing them
    in order.
 *  @param num The 64-bit number to be sent out
 */

base_text_serial& base_text_serial::operator<< (unsigned long long num)
{
    union {
        unsigned long long whole;
        unsigned char bits[8];
    } parts;
    parts.whole = num;
    *this << parts.bits[7] << parts.bits[6] << parts.bits[5] <<
        parts.bits[4]
          << parts.bits[3] << parts.bits[2] << parts.bits[1] <<
              parts.bits[0];

    return (*this);
}

#ifdef M_SQRT2 // Automatically include this code if <math.h> has been
    included

//-----------------------------------------------------------------------------
/** This method writes a single-precision floating point number to the
    serial port in
 *  exponential format (always with the 'e' notation). It calls the
    utility function
 *  __ftoa_engine, which is hiding in the AVR libraries, used by the
    Xprintf()
 *  functions when they need to convert a float into text.
 *  @param num The floating point number to be sent out
 */

base_text_serial& base_text_serial::operator<< (float num)
{
    char digits = precision;
    char buf[20];
    char* p_buf = buf;

    int exponent = __ftoa_engine ((double)num, buf, digits, 16);
    uint8_t vtype = *p_buf++;
    if (vtype & FTOA_NAN)
    {
        *this << " NaN";
        return (*this);
    }

    // Display the sign if it's negative
    if (vtype & FTOA_MINUS)
```

```
        putchar ('-');

    // Show the mantissa
    putchar (*p_buf++);
    if (digits)
        putchar ('.');
    while ((digits-- > 0) && *p_buf)
        putchar (*p_buf++);

    // Now display the exponent
    putchar ('e');
    if (exponent > 0)
        putchar ('+');
    *this << exponent;
}


//-----------------------------------------------------------------------------
/** This method writes a double-precision floating point number to the
    serial port in
 *  exponential format (always with the 'e' notation). It calls the
     utility function
 *  __ftoa_engine, which is hiding in the AVR libraries, used by the
     Xprintf()
 *  functions when they need to convert a double into text.
 *  @param num The double-precision floating point number to be sent out
 */

base_text_serial& base_text_serial::operator<< (double num)
{
    char digits = precision;
    char buf[20];
    char* p_buf = buf;

    int exponent = __ftoa_engine (num, buf, digits, 16);
    uint8_t vtype = *p_buf++;
    if (vtype & FTOA_NAN)
    {
        *this << " NaN";
        return (*this);
    }

    // Display the sign if it's negative
    if (vtype & FTOA_MINUS)
        putchar ('-');

    // Show the mantissa
    putchar (*p_buf++);
    if (digits)
        putchar ('.');
    do
```

```
        putchar (*p_buf++);
    while (--digits && *p_buf);

    // Now display the exponent
    putchar ('e');
    if (exponent > 0)
        putchar ('+');
    *this << exponent;
}

#endif // M_SQRT2


//-----------------------------------------------------------------------------
/** This function sets the global precision value, then returns a
    manipulator which
 *  causes a serial object to change its floating point precision (the
     number of digits
 *  after the decimal point which will be printed). The new precision is
     "sticky" in
 *  that its value persists until this function is called again to change
     it again.
 *  @param digits A new value for the number of digits after the decimal
     point;
 *      a maximum of 7 digits are allowed
 *  @return The serial manipulator called "manip_set_precision"
 */

ser_manipulator setprecision (unsigned char digits)
{
    if (digits > 7) digits = 7;
    bts_glob_prec = digits;

    return (manip_set_precision);
}


//-----------------------------------------------------------------------------
/** This overload allows manipulators to be used to change the base of
    displayed
 *  numbers to binary, octal, decimal, or hexadecimal. Also, and endline
     is provided
 *  with the name "endl" and the code "send_now" causes immediate
     transmission by
 *  devices which save stuff to be transmitted in buffers.
 *  @param new_manip The serial manipulator which was given
 */

base_text_serial& base_text_serial::operator<< (ser_manipulator new_manip)
{
    switch (new_manip)
```

```
    {
        case (bin):                     // Print integers in binary
            base = 2;
            break;
        case (oct):                     // Print integers in octal
            base = 8;
            break;
        case (dec):                     // Print integers in decimal
            base = 10;
            break;
        case (hex):                     // Print integers in hexadecimal
            base = 16;
            break;
        case (ascii):                   // Print chars as ASCII letters,
            etc.
            print_ascii = true;
            break;
        case (numeric):                 // Print chars as numbers
            print_ascii = false;
            break;
        case (endl):                    // Send an end-of-line
            puts (ENDL_STYLE);
            break;
        case (clrscr):                  // Send a clear-screen code
            clear_screen ();
            break;
        case (send_now):                // Send whatever's in the send
            buffer
            transmit_now ();
            break;
        case (_p_str):                  // The next string is in program
            memory
            pgm_string = true;
    };

    return (*this);
}
```

Code Block G.26: RS232 Header rs232int.h

```
//*****************************************************************************
/** \file rs232int.h
 *   This file contains a class which allows the use of a serial port on
    an AVR
 *   microcontroller. This version of the class uses the serial port
    receiver
 *   interrupt and a buffer to allow characters to be received in the
    background.
 *   The port is used in "text mode"; that is, the information which is
    sent and
 *   received is expected to be plain ASCII text, and the set of
    overloaded left-shift
 *   operators "<<" in base_text_serial.* can be used to easily send all
    sorts of data
 *   to the serial port in a manner similar to iostreams (like "cout") in
    regular C++.
 *
 *  Revised:
 *   \li 04-03-2006 JRR For updated version of compiler
 *   \li 06-10-2006 JRR Ported from C++ to C for use with some C-only
    projects; also
 *       serial_avr.h now has defines for compatibility among lots of AVR
    variants
 *   \li 08-11-2006 JRR Some bug fixes
 *   \li 03-02-2007 JRR Ported back to C++. I've had it with the
    limitations of C.
 *   \li 04-16-2007 JO Added write (unsigned long)
 *   \li 07-19-2007 JRR Changed some character return values to bool,
    added m324p
 *   \li 01-12-2008 JRR Added code for the ATmega128 using USART number 1
    only
 *   \li 02-14-2008 JRR Split between base_text_serial and rs232 files
 *   \li 05-31-2008 JRR Changed baud calculations to use CPU_FREQ_MHz from
    Makefile
 *   \li 06-01-2008 JRR Added getch_tout() because it's needed by 9Xstream
    modems
 *   \li 07-05-2008 JRR Changed from 1 to 2 stop bits to placate finicky
    receivers
 *   \li 12-22-2008 JRR Split off stuff in base232.h for efficiency
 *   \li 06-30-2009 JRR Received data interrupt and buffer added
 *
 *  License:
 *     This file is released under the Lesser GNU Public License, version
    2. This
 *     program is intended for educational use only, but it is not limited
    thereto.
 */
//*****************************************************************************

/// This define prevents this .h file from being included more than once
```

```
    in a .cc file
#ifndef _RS232_H_
#define _RS232_H_

#include <avr/interrupt.h>              // Header for AVR interrupt
    programming
#include "base232.h"                    // Grab the base RS232-style
    header file
#include "base_text_serial.h"           // Pull in the base class header
    file


// If the chip for which we are compiling has dual serial ports, define
    two character
// received interrupts, one for each port. This is for ATmega128 and 324P
    and similar
// chips. This section may need to be expanded to work with some other
    chips
#if defined UCSR0A
    #define RSI_CHAR_RECV_INT_0 USART0_RX_vect
    #define RSI_CHAR_RECV_INT_1 USART1_RX_vect
// There's no second serial port, so define one port's interrupt. This is
    for ATmega8
// and ATmega32 and similar chips. This section will need to be expanded
    if other
// single-UART/USART chips are used, as the vector may have a different
    name
#else
    #define RSI_CHAR_RECV_INT_0 USART_RXC_vect
#endif


/// This is the size of the buffer which holds characters received by the
    serial port.
#define RSINT_BUF_SIZE   128


//-----------------------------------------------------------------------------
/** This class controls a UART (Universal Asynchronous Receiver
    Transmitter), a common
 *  serial interface. It talks to old-style RS232 serial ports (through a
     voltage
 *  converter chip such as a MAX232) or through a USB to serial converter
     such as a
 *  FT232RL chip. The UART is also sometimes used to communicate directly
     with other
 *  microcontrollers, sensors, or wireless modems.
 */

class rs232 : public base_text_serial, public base232
{
```

```
    // This protected data can only be accessed from this class or its
       descendents
    protected:
        uint8_t port_num;               ///< The USART number, 0 or 1

    // Public methods can be called from anywhere in the program where
       there is a
    // pointer or reference to an object of this class
    public:
        // The constructor sets up the UART, saving its baud rate and port
           number
        rs232 (unsigned int = 9600, unsigned char = 0);

        /// This method writes one character to the serial port.
        bool putchar (char);

        void puts (char const*);        // Write a string constant to
           serial port
        bool check_for_char (void);     // Check if a character is in the
           buffer
        char getchar (void);            // Get a character; wait if none
           is ready
        void clear_screen (void);       // Send the 'clear display screen'
           code
//      char getch_tout (unsigned int);  // Try a given number of times to
    get char
};

#endif // _RS232_H_
```

Code Block G.27: RS232 Class rs232int.cpp

```
//*****************************************************************************
/** \file rs232int.cpp
 *   This file contains a class which allows the use of a serial port on
    an AVR
 *   microcontroller. This version of the class uses the serial port
    receiver
 *   interrupt and a buffer to allow characters to be received in the
    background.
 *   The port is used in "text mode"; that is, the information which is
    sent and
 *   received is expected to be plain ASCII text, and the set of
    overloaded left-shift
 *   operators "<<" in base_text_serial.* can be used to easily send all
    sorts of data
 *   to the serial port in a manner similar to iostreams (like "cout") in
    regular C++.
 *
 *  Revised:
 *   \li 04-03-2006 JRR For updated version of compiler
 *   \li 06-10-2006 JRR Ported from C++ to C for use with some C-only
    projects; also
 *       serial_avr.h now has defines for compatibility among lots of AVR
    variants
 *   \li 08-11-2006 JRR Some bug fixes
 *   \li 03-02-2007 JRR Ported back to C++. I've had it with the
    limitations of C.
 *   \li 04-16-2007 JO Added write (unsigned long)
 *   \li 07-19-2007 JRR Changed some character return values to bool,
    added m324p
 *   \li 01-12-2008 JRR Added code for the ATmega128 using USART number 1
    only
 *   \li 02-14-2008 JRR Split between base_text_serial and rs232 files
 *   \li 05-31-2008 JRR Changed baud calculations to use CPU_FREQ_MHz from
    Makefile
 *   \li 06-01-2008 JRR Added getch_tout() because it's needed by 9Xstream
    modems
 *   \li 07-05-2008 JRR Changed from 1 to 2 stop bits to placate finicky
    receivers
 *   \li 12-22-2008 JRR Split off stuff in base232.h for efficiency
 *   \li 06-30-2009 JRR Received data interrupt and buffer added
 *
 *  License:
 *     This file is released under the Lesser GNU Public License, version
    2. This
 *     program is intended for educational use only, but it is not limited
    thereto.
 */
//*****************************************************************************

#include <stdint.h>
```

```
#include <stdlib.h>
#include <avr/io.h>
#include "rs232int.h"


// Every AVR has at least one serial port, so enable at least one receiver
   buffer
/// This buffer holds characters received through serial port 0 by the ISR.
uint8_t* rcv0_buffer = NULL;

/// This index is used to write into serial character receiver buffer 0.
uint16_t rcv0_read_index;

/// This index is used to read from serial character receiver buffer 0.
uint16_t rcv0_write_index;

// If there's a UCSR0A register, there are 2 serial ports, so enable
   another buffer
#ifdef UCSR1A
    /// This buffer holds characters received through serial port 1 by the
       ISR.
    uint8_t* rcv1_buffer = NULL;

    /// This index is used to write into serial character receiver buffer
       1.
    uint16_t rcv1_read_index;

    /// This index is used to read from serial character receiver buffer 1.
    uint16_t rcv1_write_index;
#endif


//-------------------------------------------------------------------------------
/** This method sets up the AVR UART for communications. It calls the
   base_text_serial
 *  constructor, which prepares to convert numbers to text strings, and
    the base232
 *  constructor, which does the work of setting up the serial port. Note
    that the user
 *  program must call sei() somewhere to enable global interrupts so that
    this driver
 *  will work. It is not called in this constructor because it's common to
    construct
 *  many drivers which use interrupts, including this one, and then enable
    interrupts
 *  globally using sei() after all the constructors have been called.
 *  @param baud_rate The desired baud rate for serial communications.
    Default is 9600
 *  @param port_number The number of the serial port, 0 or 1 (the second
    port numbered
 *                   1 only exists on some processors). The default is
```

```
    port 0
 */

rs232::rs232 (unsigned int baud_rate, unsigned char port_number)
    : base_text_serial (), base232 (baud_rate, port_number)
{
    // Save the number of the serial port, 0 or 1
    port_num = port_number;

    // If we're compiling for a chip with UCSR0A defined, it has dual
        serial ports
    // (examples are ATmega324P and ATmega128). Set up Port 0 or Port 1
    #if defined UCSR0A // Serial port number 0
        if (port_number == 0)
        {
            UCSR0B |= (1 << RXCIE0);     // Receive complete interrupt
                enable

            // Allocate some memory for the receiver buffer and reset the
                indices
            rcv0_buffer = new uint8_t[RSINT_BUF_SIZE];
            rcv0_read_index = 0;
            rcv0_write_index = 0;
        }
        else // Serial port number 1
        {
        #if defined UCSR1A
            UCSR1B |= (1 << RXCIE1);     // Receive complete interrupt
                enable

            // Allocate some memory for the receiver buffer and reset the
                indices
            rcv1_buffer = new uint8_t[RSINT_BUF_SIZE];
            rcv1_read_index = 0;
            rcv1_write_index = 0;
        #endif // UCSR1A
        }
    // We're compiling for a chip which doesn't define UCSR0A; assume it
        has only one
    // serial port.
    #else
        UCSRB |= (1 << RXCIE);           // Receive complete interrupt
            enable

        // Allocate some memory for the receiver buffer and reset the
            indices
        rcv0_buffer = new uint8_t[RSINT_BUF_SIZE];
        rcv0_read_index = 0;
        rcv0_write_index = 0;
    #endif
```

```
    // The Xiphos 1.0 board may need the pullup activated on the RXD1 line
       in order to
    // use the XBee radio module
    #ifdef XIPHOS_HACKS
        if (port_number == 1)
            PORTD |= 0x04;
    #endif
}


//-------------------------------------------------------------------------------
/** This method sends one character to the serial port. It waits until the
    port is
 *  ready, so it can hold up the system for a while. It times out if it
     waits too
 *  long to send the character; you can check the return value to see if
     the character
 *  was successfully sent, or just cross your fingers and ignore the
     return value.
 *  Note 1: It's possible that at slower baud rates and/or higher
     processor speeds,
 *  this routine might time out even when the port is working fine. A
     solution would
 *  be to change the count variable to an integer and use a larger
     starting number.
 *  Note 2: Fixed! The count is now an integer and it works at lower baud
     rates.
 *  @param chout The character to be sent out
 *  @return True if everything was OK and false if there was a timeout
 */

bool rs232::putchar (char chout)
{
    // Now wait for the serial port transmitter buffer to be empty
    for (unsigned int count = 0; ((*p_USR & mask_UDRE) == 0); count++)
    {
        if (count > UART_TX_TOUT)
            return (false);
    }

    // Clear the TXCn bit so it can be used to check if the serial port is
       busy. This
    // check needs to be done prior to putting the processor into sleep
       mode. Oddly,
    // the TXCn bit is cleared by writing a one to its bit location
    *p_USR |= mask_TXC;

    // The CTS line is 0 and the transmitter buffer is empty, so send the
       character
    *p_UDR = chout;
    return (true);
```

```
}


//---------------------------------------------------------------------------
/** This method writes all the characters in a string until it gets to the
    '\\0' at
 *  the end. Warning: This function blocks until it's finished.
 *  @param str The string to be written
 */

void rs232::puts (char const* str)
{
    while (*str) putchar (*str++);
}


//---------------------------------------------------------------------------
/** This method gets one character from the serial port, if one is there.
    If not, it
 *  waits until there is a character available. This can sometimes take a
     long time
 *  (even forever), so use this function carefully. One should almost
     always use
 *  check_for_char() to ensure that there's data available first.
 *  @return The character which was found in the serial port receive buffer
 */

char rs232::getchar (void)
{
    uint8_t recv_char;                  // Character read from the queue

    // Wait until there's a character in the receiver queue
    #ifdef UCSR0A // If this is a dual-port chip
        if (port_num == 0)
        {
            while (rcv0_read_index == rcv0_write_index);
            recv_char = rcv0_buffer[rcv0_read_index];
            if (++rcv0_read_index >= RSINT_BUF_SIZE)
                rcv0_read_index = 0;
        }
        else // This is port 1 of a dual-port chip
        {
        #if defined UCSR1A
            while (rcv1_read_index == rcv1_write_index);
            recv_char = rcv1_buffer[rcv1_read_index];
            if (++rcv1_read_index >= RSINT_BUF_SIZE)
                rcv1_read_index = 0;
        #endif // UCSR1A
        }
    #else // This chip has only one serial port
        while (rcv0_read_index == rcv0_write_index);
```

```
        recv_char = rcv0_buffer[rcv0_read_index];
        if (++rcv0_read_index >= RSINT_BUF_SIZE)
            rcv0_read_index = 0;
    #endif

    return (recv_char);
}


//-----------------------------------------------------------------------------
/** This method checks if there is a character in the serial port's
    receiver queue.
 *  The queue will have been filled if a character came in through the
     serial port and
 *  caused an interrupt.
 *  It returns 1 if there's a character available, and 0 if not.
 *  @return True for character available, false for no character available
 */

bool rs232::check_for_char (void)
{
    #ifdef UCSR1A                       // If this is a dual-port chip
        if (port_num == 0)
            return (rcv0_read_index != rcv0_write_index);
        else
            return (rcv1_read_index != rcv1_write_index);
    #else                               // This chip has only one serial
        port
        return (rcv0_read_index != rcv0_write_index);
    #endif
}


//-----------------------------------------------------------------------------
/** This method sends the ASCII code to clear a display screen. It is
    called when the
 *  format modifier 'clrscr' is inserted in a line of "<<" stuff.
 */

void rs232::clear_screen (void)
{
    putchar (CLRSCR_STYLE);
}


//-----------------------------------------------------------------------------
/** \cond NOT_ENABLED (This ISR is not to be documented by Doxygen)
 *  This interrupt service routine runs whenever a character has been
     received by the
 *  first serial port (number 0). It saves that character into the
     receiver buffer.
```

```
 */

ISR (RSI_CHAR_RECV_INT_0)
{
    // When this ISR is triggered, there's a character waiting in the
       USART data reg-
    // ister, and the write index indexes the place where that character
       should go

    #if defined UCSR0A // If this is a dual-serial-port chip (ATmega324P,
       128, etc.)
        rcv0_buffer[rcv0_write_index] = UDR0;
    #else // If this chip has only a single serial port (ATmega8, 32, etc.)
        rcv0_buffer[rcv0_write_index] = UDR;
    #endif

    // Increment the write pointer
    if (++rcv0_write_index >= RSINT_BUF_SIZE)
        rcv0_write_index = 0;

    // If the write pointer is now equal to the read pointer, that means
       we've just
    // overwritten the oldest data. Increment the read pointer so that it
       doesn't seem
    // as if the buffer is empty
    if (rcv0_write_index == rcv0_read_index)
        if (++rcv0_read_index >= RSINT_BUF_SIZE)
            rcv0_read_index = 0;
}


#ifdef UCSR1A // The second ISR is only compiled for processors with dual
    serial ports
    //-------------------------------------------------------------------------
    /** This interrupt service routine runs whenever a character has been
       received by the
    *  first serial port (number 0). It saves that character into the
       receiver buffer.
    */

    ISR (RSI_CHAR_RECV_INT_1)
    {
        // Read the character from the serial port receiver buffer
        rcv1_buffer[rcv1_write_index] = UDR1;

        // Increment the write pointer
        if (++rcv1_write_index >= RSINT_BUF_SIZE)
            rcv1_write_index = 0;

        // If the write pointer is now equal to the read pointer, that
           means we've just
```

```
        // overwritten the oldest data. Increment the read pointer so that
            it doesn't seem
        // as if the buffer is empty
        if (rcv1_write_index == rcv1_read_index)
            if (++rcv1_read_index >= RSINT_BUF_SIZE)
                rcv1_read_index = 0;
    }
#endif // Dual serial ports
/** \endcond (End of section which is not to be documented by Doxygen) */
```